Tampereen yliopisto

Henri Vekuna

PÄÄSYNHALLINTA WEB-SOVELLUSKEHYKSISSÄ

Informaatioteknologian ja viestinnän tiedekunta
Pro gradu -tutkielma
Maaliskuu 2019

TIIVISTELMÄ

Henri Vekuna: Pääsynhallinta web-sovelluskehyksissä
Pro gradu -tutkielma, 53 sivua
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Maaliskuu 2019

Pääsynhallinta eli auktorisointi on keskeinen osa web-sovelluskehysten käyttäjänhallintaa. Se mahdollistaa käyttäjän pääsyn estämisen tai sallimisen web-sovelluksen eri osiin sekä sivun tiettyjen visuaalisten elementtien piilottamisen käyttäjältä. Pääsynhallintaa tarvitaan sellaisissa web-sovelluksissa, joiden sisältöä halutaan rajata käyttäjän ominaisuuksien, kuten sisäänkirjautuneisuuden tai käyttäjäroolin perusteella.

Ohjelmistokehitys painottuu entistä enemmän internetiin ja web-sivujen määrä kasvaa koko ajan. Web-sovelluskehykset ovat tärkeä osa web-kehitystä. Web-sovelluskehys on jollain ohjelmointikielellä toteutettu kokoelma web-sovelluksissa yleisesti tarvittavia toimintoja. Sovelluskehys toimii runkona, jonka päälle monimutkaiset sovellukset rakennetaan.

Tässä tutkielmassa tarkastellaan, miten pääsynhallinta toimii eri web-sovelluskehyksissä ohjelmointiteknisestä näkökulmasta. Tutkielmassa kartoitetaan ja vertaillaan eri pääsynhallintakirjastojen ominaisuuksia. Kolmen kehyksen auktorisointia käsitellään tarkemmin esimerkkiohjelmien avulla.

Pääsynhallinnan keskeisimpinä tyyppeinä roolit ja attribuutit löytyvät kaikista web-sovelluskehyksistä. Useimmissa kehyksistä löytyy mahdollisuus rajata pääsyä resurssikohtaisesti, minkä voi saada helpostikin käyttöön esimerkiksi metodin lisäparametrilla. Koodissa sisäänpääsyn tarkastaminen tapahtuu oliosta saadun totuusarvon perusteella joko ehtolauseissa tai koristajissa sekä mallineiden puolella html-tageja muistuttavissa kontrollirakenteissa. Pääsynhallinta voi kuulua sovelluskehyksen ydintoimintoihin tai olla erikseen ladattava kirjasto. Sääntöjen monipuolisuus ja tallennustapa vaihtelevat kirjastoittain.

Avainsanat: **auktorisointi**, **pääsynhallinta**, **web-sovelluskehykset**

1 Johdanto

Auktorisointi eli pääsynhallinta on web-sovelluskehysten keskeinen ominaisuus ja osa web-kehysten käyttäjänhallintaa tunnistautumisen ohella. Pääsynhallintaa tarvitaan kaikilla niillä dynaamisilla web-sivuilla, joissa käyttäjän tulee olla tunnistettu ja käyttäjän toimia halutaan rajoittaa.

Tutkielmassa tarkastellaan, miten auktorisointi toimii web-sovelluskehyksissä. Eri web-sovelluskehykset tarjoavat erilaisia mekanismeja pääsynhallinnan toteuttamiseen, joiden toimintatapoja tutkielma käsittelee. Näkökulma on ohjelmointitekninen.

Pääsynhallinta on tilannekohtaista eli riippuu täysin sovelluksen asettamista vaatimuksista. Tämän vuoksi pääsynhallinnan toteutus ja sopivan auktorisoinnin valitseminen riippuu sovelluksen tarpeista, jolloin valmiita, yleispäteviä ja kaikkiin tilanteisiin sopivia ratkaisuja ei ole. Ohjelmoija soveltaa kehysten pääsynhallintamekanismeja tapauskohtaisesti.

Luvussa 2 käsitellään auktorisointia yleisellä ja teoreettisella tasolla. Siinä määritellään ensin auktorisointi käsitteenä ja käydään läpi hyvän pääsynhallinnan turvallisuusvaatimuksia ja ominaisuuksia. Lopuksi esitellään pääsynhallinnan eri tyyppejä historiallisesta näkökulmasta. Luvussa 3 määritellään sovelluskehykset yleisesti. Luvussa 4 käsitellään tarkemmin web-sovelluskehyksiä ja niiden yhteydessä käydään läpi web-sivujen ja internetin toimintaa yleisesti, sessiota, autentikointia sekä MVC-mallia. Luku 5 kuvaa pääsynhallinnan toimintamekanismeja sekä suunnittelumalleja. Siihen on valittu erilaisia ja yleisiä web-kehyksiä, joiden osalta kuvataan käytännössä. miten auktorisointi niissä toimii. Luvussa 6 käsitellään sovelluskehyksillä Node.js, Django ja ASP.Net tehtyjä esimerkkiohjelmia, joiden avulla tarkastellaan kyseisten kehysten auktorisointia tarkemmin.

2 Auktorisointi

Sanakirjassa auktorisointi eli pääsynhallinta määritellään prosessiksi, jossa subjekti pääsee tai ei pääse tiettyyn resurssiin (objekti) oikeuksiensa perusteella [Butterfield and Ngondi, 2016]. Subjekti on jokin entiteetti, joka käyttää objektia jollain tavalla. Käytännössä subjekti on usein ohjelma tai prosessi, joka toimii käyttäjän puolesta, jolloin käyttäjä ja subjekti ovat teknisesti eri asioita [Ferraiolo et al., 2007].

Objekti eli resurssi on pääsynhallinnan kohde, esimerkiksi web-sivun tai ohjelman osa, jokin tiedosto tai kansio. Subjekti tekee objektille jonkin operaation, se lukee objektia tai näkee objektin, kirjoittaa tai tallentaa objektiin tietoa tai käyttää sitä jollain muulla tavalla. Perinteisesti objektit on nähty Ferraiolon mukaan passiivisiksi entiteeteiksi, koska niitä tulee suojella ja ne sisältävät tietoa, mutta käytännössä objekti voi olla itsekin aktiivinen toimija [Ferraiolo et al., 2007].

Oikeus on käyttäjän lupa tehdä operaatio objektille. Politiikka (*policy*) eli käytäntö on sarja sääntöjä, lupia tai suhteita, jotka määrittelevät pääsyoikeudet. Paikoissa, jossa auktorisointia käytetään, tarkistetaan subjektin pääsyoikeus ja subjekti päästetään objektiin, jos subjektilla on siihen oikeus ja ei päästetä, mikäli subjektilla ei ole oikeutta [Butterfield and Ngondi, 2016]. Pääsyoikeuksien tarkistaminen on käytännön toteutusta, joiden tulee noudattaa järjestelmälle määriteltyä politiikkaa.

Auktorisointi on eri prosessi kuin autentikointi eli tunnistautuminen. Tunnistautuminen täytyy tehdä ennen auktorisointia. Esimerkiksi blogisovelluksen politiikka voisi olla, että käyttäjä voi muokata vain omia kirjoituksiaan. Käyttäjän tulee olla tässä vaiheessa jo kirjautunut sisään, jotta voidaan tarkastaa, onko käyttäjä sama kuin kirjoituksen tekijä. Oikeuksien tarkistaminen tapahtuu siinä osassa ohjelmaa, mistä pääsee muokkaamaan omia kirjoituksiaan.

Pääsynhallintaa on myös sähköisen maailman ulkopuolella, kuten tapahtumissa, joihin ostetaan pääsylippu, jolla pääsee tapahtumaan. Seuraavassa kohdassa käsitellään pääsynhallinnan ja yleisen tietoturvan kolmea vaatimusta.

2.1 Turvallisuusvaatimukset

Auktorisoinnin ja muun tietoturvan turvallisuusvaatimukset ovat luottamuksellisuus (*confidentiality*), saatavuus (*availability*) ja eheys (*integrity*) [Alberts and Dorofee, 2004]. Luottamuksellisuus tarkoittaa objektien suojaamista luvattomalta käytöltä eli pääsyltä, muokkaamiselta ja poistamiselta. Kun pääsynhallinta on luottamuksellinen, luvattomien käyttäjien ei anneta vahingossa tai tahallaan käyttää objekteja.

Saatavuus tarkoittaa pääsynhallinnan toimimista oikein aina sitä tarvittaessa. Saatavuutta huonontavat huoltotoimenpiteet ja virheet, joiden takia järjestelmä ei ole käytössä tai toimi oikein kaikkina ajan hetkinä.

Eheydellä tarkoitetaan objektien ja mekanismien pysyvyyttä ja muuttumattomuutta. Sovelluksen auktorisoinnissa ei saa olla sellaisia virheitä, jotka mahdollistavat objektien

tai pääsynhallinnan logiikan muuttumisen itsekseen tai luvattomasti. Luvan saaneet käyttäjät voivat toki muuttaa objekteja, mutta muuten niiden sisältämät tiedot pitävät aina paikkansa. Seuraavassa kohdassa esitellään hyvän pääsynhallinnan suunnitteluperiaatteita, joilla pyritään turvallisuusvaatimusten toteutumiseen.

2.2 Pääsynhallinnan suunnitteluperiaatteita

Saltzer ja Schroeder [1975] esittelivät tietoturvan suunnitteluperiaatteita, joista osa on alun perin kryptografian periaatteita. Suunnitteluperiaatteet auktorisoinnin kontekstiin ovat koonneet Ferraiolo ja muut [2007].

Tehtävien hajauttaminen (*Separation of duties, SoD*) tarkoittaa kriittisen, useita vaiheita sisältävän, tehtävän jakamista useammalle kuin yhdelle ihmiselle turvallisuussyistä. Kun vähintään kaksi ihmistä on vastuussa velvollisuudesta, se vähentää virheiden tai vahingon mahdollisuutta ja luo sisäistä kontrollia [Ferraiolo et al., 2007]. Tehtävien hajauttaminen on perusperiaate tärkeissä, turvallisuutta ja oikeellisuutta vaativissa asioissa ja tunnettu varsinkin pankkialalla [Zhang, 2010]. Hajauttaminen voi tarkoittaa myös kahden tai useamman ehdon toteutumista ennen kuin pääsy sallitaan [Salzer and Schroeder, 1975].

Vähimpien oikeuksien periaate (*Princible of least priviledge*) tarkoittaa, että komponenteilla (subjekteilla) on vain ne pääsyoikeudet, joita tarvitaan [Ince, 2013]. Käyttäjillä tulisi olla minimimäärä oikeuksia eli vain sen verran kuin käyttäjän kuuluu saada kyseisen käyttäjän tai käyttäjäryhmän tehtäviä varten. Ylimääräisten oikeuksien väärinkäyttäminen voisi vahingoittaa järjestelmää.

Vikaturvallisuus (*Fail-safe*) tarkoittaa yleisesti järjestelmän rakentamista siten, että virheen sattuessa minimoidaan tai vältetään vahinko [Butterfield and Ngondi, 2016]. Auktorisoinnissa oletuksena pääsy evätään ja pääsy sallitaan erikseen säännöillä [Ferraiolo et al., 2007]. Tällöin virheen sattuessa sallivat säännöt lakkaisivat toimimasta, jolloin ketään ei varmuuden vuoksi päästetä käsiksi objektiin, ei normaalisti sallittuja subjekteja eikä varsinkaan kiellettyjä. Jos pääsy toisin päin sallittaisiin oletuksena ja evättäisiin erikseen, virheen sattuessa kuka tahansa voisi päästä käsiksi objekteihin, mikä on turvatonta ja epäluottamuksellista.

Taloudellisesti toteutettu järjestelmä (*Economy of mechanism*) tarkoittaa, että järjestelmä on yksinkertainen ja pieni, vähän resursseja vievä. Yksinkertaista järjestelmää voidaan myöhemmin analysoida helpommin kuin monimutkaista ja todeta sen toimivuus. Yksinkertainen järjestelmä vie vähemmän tilaa, on jo tekovaiheessa helpompi rakentaa kuin monimutkainen ja siihen tulee epätodennäköisemmin virheitä [Salzer and Schroeder, 1975].

Avoin rakenne (*Open design*) tarkoittaa, että pääsynhallinnan toimintamekanismi ei ole salainen, vaan muut voivat nähdä miten se toimii. Se lisää skeptisten käyttäjien luottamusta ja kuka tahansa voi havaita siitä epäkohtia. Laajassa käytössä olevan järjestelmän

toimintamekanismin salassapito voi olla täysin mahdotontakin [Salzer and Schroeder, 1975]. Rakenteen avoimuus on Auguste Kerckhoffin keksimä kryptografian käsite [Mrdovic and Perunicic, 2008]. Pääsynhallinnan toimivuuden ei pitäisi olla riippuvaista sen rakenteen avoimuudesta. Vaikka konstruktio on avoin, itse käyttäjän tunnistautumisen tarvittavat tiedot ovat salaisia, kuten salasanat ja session tunnukset.

Täydellinen tarkistus (*Complete mediation*) tarkoittaa, että jokainen subjektin pyyntö päästä käsiksi objektiin tarkistetaan joka kerta täydellisesti ja pääsy sallitaan tai evätään aina erikseen [Salzer and Schroeder, 1975]. Jos subjektin oikeudet muuttuvat pääsy-yritysten välillä, järjestelmä tekee joka kerralla nykyisiä oikeuksia vastaavan päätöksen. Tällä varmistetaan, ettei päätös ole automaattisesti sama kuin ennenkin, jolloin edellisiä päätöksiä sisäänpääsystä ei oteta huomioon. Täydellinen tarkistus turvaa järjestelmän eheyden pääsyoikeuspolitiikan muuttuessa. Toisaalta oikeuksien tarkistaminen liian usein voi viedä resursseja, ja ainakin saman session sisällä oikeuksien tarkistaminen useammin kuin kerran voi olla turhaa.

Käytettävyys (*Psychological acceptance*) tarkoittaa, että pääsynhallinnan tulee vastata ihmisen sisäisiä malleja ja uskomuksia, jotta ihmiset käyttäisivät sitä oikein [Salzer and Schroeder, 1975]. Käytettävyys vähentää käyttäjän tekemiä virheitä ja parantaa käyttökokemusta, mikä tekee käyttäjästä tyytyväisemmän järjestelmään.

Vähimpien yhteisten mekanismien periaate (*Least common mechanism*) tarkoittaa, että usean käyttäjän välisten jaettujen mekanismien määrä tulee minimoida. Kun eri käyttäjillä on mahdollisimman vähän yhteisiä toimintoja ja informaatiota, se vähentää väärinkäytön mahdollisuutta. Mekanismi tarkoittaa tietynlaista pääsytapaa objektiin ja mekanismien rajoittaminen vähentää tärkeän informaation leviämistä. Kahdella käyttäjällä ei tulisi olla yhteistä pääsyoikeusinstanssia objektiin. Kun mahdollisimman vähän subjekteja pääsee käyttämään objektia tietyllä tavalla, se vähentää todennäköisyyttä objektin sisältämän tiedon väärinkäyttöön tai itse pääsyväylän sabotointiin [Salzer and Schroeder, 1975; Gegick and Barnum, 2013].

2.3 Harkinnanvarainen ja pakollinen pääsynhallinta (DAC ja MAC)

Harkinnanvaraisessa pääsynvalvonnassa (*Discretionary access control, DAC*) subjektit päättävät oikeuksien asettamisesta. Subjekteja, jotka voivat alun perin antaa oikeuksia kutsutaan harkinnanvaraisen pääsynhallinnan yhteydessä omistajiksi (*owner*), esimerkiksi tiedoston tekijä on sen omistaja. [Butterfield and Ngondi, 2016]

Omistajat voivat vapaasti päättää mitä oikeuksia ja keille käyttäjille he antavat, lisäksi oikeuksia omistajilta saaneet käyttäjät voivat antaa oikeuksia edelleen eteenpäin. Kun lupia tiedostoihin voi antaa toisille, DAC on altis haittaohjelmille ja toisten tiedostojen informaation leviämiselle [Ferraiolo et al., 2007]. Harkinnanvarainen pääsynhallinta ei ole jäykkä järjestelmä. Kun vapaus pääsyoikeuksien asettamisesta on subjekteilla, järjestelmä on joustava, mutta kääntöpuolena on tietoturvauhkia.

DAC on käytössä käyttäjäryhmittäin Linuxin ja muiden Unix-pohjaisten käyttöjärjestelmien tiedostoissa ja kansioissa. Oikeuksia on kolme: luku (*r*), kirjoitus (*w*) ja suoritus (*x*). Kirjoitusoikeudella yksittäistä tiedostoa tai kansion tiedostoja voi muokata, kirjoittaa ja poistaa. Kansioissa lukemisoikeudella saadaan sen tiedostojen pelkät nimet näkyviin, mutta tiedostojen varsinainen tarkastelu ja liitännäistietojen (koko jne.) näkyminen vaativat suoritusoikeuden. Luokkia eli käyttäjäryhmiä on kolme: tiedoston omistaja (*owner*), ryhmä (*group*) ja muut käyttäjät (*others*). Tiedoston omistaja voi muuttaa oikeuksia *chmod*-komennolla, mikä tekee auktorisoinnista harkinnanvaraisen [Kothari, 2015].

Yhteisöpalvelu Facebookissa henkilö voi määrittää itse ketkä näkevät hänen asettamia tietojaan profiilissa, kuten sähköpostia, puhelinnumeroa, työ- ja asuinpaikkaa, kouluja ja perheenjäseniä. Jokaiselle yksittäiselle tiedolle voi asettaa näkyvyyden erikseen yleisönvalitsimella [Facebook, 2018]. Tiedot voivat olla kaikille julkisia, näkyä vain kavereille tai itselle, kavereiden kavereille ja yksittäisiä henkilöitä voi sisällyttää tai poissulkea. Henkilö on oman tietosivunsa omistaja.

Pakollisessa pääsynhallinnassa (*Mandatory access control, MAC*) ylläpito asettaa pääsyoikeudet, eivätkä käyttäjät saa antaa oikeuksia vahingossa tai tarkoituksella, toisin kuin harkinnanvaraisessa pääsynhallinnassa [Zhang, 2010]. Sen pääsynhallintaa ohjataan keskitetysti ja koko systeemin tasolla.

Pakollisen pääsyhallinnan pääsypolitiikka on ahtaasti määriteltyä ja rajoittavaa. Siinä on hierarkkisia turvatasoja, jotka ovat toistensa poissulkevia siten, että ylemmät tasot dominoivat alempia. Käyttäjälle on annettu joku turvataso, joka sisältää mahdolliset alemmat tasot ja niiden oikeudet, mutta ylempien tasojen oikeuksia ei voi saada. Käyttäjä ei myöskään voi siirtyä tasojen välillä. Pakollinen pääsynhallinta on jäykempi ja tiukempi järjestelmä kuin DAC.

Pakollinen pääsynhallinta tulee kyseeseen, jos tarvitaan hyvin korkeaa tietoturvaa ja keskitetysti ohjattua hierarkkista rakennetta. Tällainen voisi olla jokin osa armeijan tietojärjestelmää, jonka sisältämien dokumenttien turvatasoina olisi esimerkiksi julkinen, salainen ja huippusalainen. Armeijan hierarkiassa ylimmät ja erityistehtävissä palvelevat pääsisivät korkeamman turvatason objekteihin kuin alemmat.

Pakollista ja harkinnanvaraista pääsynhallintaa käsitellään yhdessä, koska molemmat esiteltiin alun perin Yhdysvaltain Puolustusministeriön julkaisemassa *Trusted Computer System Evaluation Criteria* -artikkelissa [Latham, 1986]. Pääsynhallintaa oli aluksi juuri sotilaskäytössä ja se laajentui myöhemmin kaupallisiin tietojärjestelmiin [Lipner 2015]. Alun perin pääsynhallintaa oli myös yliopistojen tietoverkoissa, mihin tarkoitukseen kehitettiin tietorakennepohjaiset pääsynhallintamatriisi ja -lista.

2.4 Pääsynhallintamatriisi ja pääsynhallintalista

Pääsynhallintamatriisi (*Access control matrix*) on Lampsonin [1974] esittämä tietorakenneratkaisu auktorisointiin, jossa toimialueet (subjektit) ja objektit ovat matriisissa

(*i,j*). Matriisin riveinä ovat subjektit ja sarakkeina objektit ja yksittäisen matriisin alkiona lista luvista. Objekteista ensimmäisinä ovat subjektit itse, siis toimialueiden oikeudet itseensä ja muihin toimialueihin.

Pääsynhallintamatriisi on pelkästään teoreettinen malli ja ei sellaisenaan käytössä. Jos subjekteja on paljon, matriisista tulee iso, se veisi Zhangin ja muiden [2010] mukaan nykyisillä laajoilla web-sovelluksilla paljon tilaa. Matriisista tulee harva, koska kaikille subjekteille *i* ei ole annettu lupia kaikkiin objekteihin *j*, jolloin kyseiset alkiot (*i*, *j*) ovat tyhjiä. Harva matriisi tuhlaa tilaa, mikä mainitaan jo alkuperäisessä artikkelissa [Lampson, 1974; Sandhu and Samarati, 1994].

Pääsynhallintamatriisi on yksi ensimmäisistä formaaleista auktorisoinnin määritelmistä ja siten historiallisesti tärkeä. Se sisältää ensimmäisenä subjektin ja objektin käsitteet. Artikkelin kieliasu perustuu 1970-luvun keskustietokoneiden osituskäyttöön ja tietorakenteena on yksinkertainen matematiikasta periytyvä matriisi.

Pääsynhallintalista (*Access control list, ACL*) on pääsynhallintamatriisin toteutus, jossa objektilla on lista niistä subjekteista, joilla on lupa objektiin. Lista vastaa pääsynhallintamatriisin saraketta. Yhden tiedoston pääsynhallintalista sisältää niiden käyttäjien tai käyttäjäryhmien nimet, joilla on oikeus suorittaa tiedosto ja kun lista on tyhjä, kenelläkään ei ole oikeuksia kyseiseen tiedostoon [Sandhu and Samarati, 1994].

Kun subjekti pyytää saada käyttää tiedostoa, järjestelmä käy pääsynhallintalistan läpi ja antaa luvan, jos subjekti löytyy listasta. Matriisiin verrattuna lista säästää tilaa, koska lista sisältää vain toteutuneet oikeudet ja matriisin turhia tyhjiä alkioita ei ole [Zhang, 2010].

Pääsynhallintalistasta näkee suoraan ketkä saavat käyttää tiedostoa, mutta toisin päin eli mihin tiedostoihin subjektilla on oikeus ei näe yhtä helposti. Yhden käyttäjän kaikkien oikeuksien selvittämiseksi tulisi käydä läpi kaikki tiedostot, mikä vie aikaa. Toiseen suuntaan toimiva lista on oikeuslista (c*apability list)*, joka sisältää subjektin kaikki oikeudet tiedostoihin ja kansioihin, mikä vastaa matriisin tarkastelua riveittäin. [Sandhu and Samarati, 1994]

Toisin kuin abstrakti pääsynhallintamatriisi, pääsynhallintalista on käyttökelpoinen, sitä käytetään esimerkiksi reitittimissä tietoliikenteen kontrollointiin. Laitteella on yksi lista per tiedonsiirtoprotokolla ja reitittimen listassa joko sallitaan tai estetään tietyt osoitteet [Belosa, 2015]. Linux-käyttöjärjestelmässä tiedostoilla ja kansioilla voi olla ryhmäkohtaisten käyttöoikeuksien lisäksi erillinen lista yksittäisistä käyttäjistä ja heidän oikeuksistaan. Esimerkiksi listan arvo "*user:henri:r-x"* kertoo, että käyttäjällä henri on luku- ja suoritusoikeudet, muttei kirjoitusoikeutta [Kerrisk, 2010].

Pääsynhallintalistaa voi käyttää taustalla toimivana tietorakenteena web-sovelluksissä, joissa on tarve tarpeeksi yksinkertaiselle objektikohtaiselle auktorisoinnille, jota ei tarkastella subjektikohtaisesti.

2.5 Roolipohjainen auktorisointi

Roolipohjaisen auktorisoinnin perusperiaatteena käyttäjät kuuluvat rooleihin ja rooleilla on oikeuksia. Käyttöoikeuksia ei siis ole annettu suoraan yksittäisille käyttäjille, vaan välissä on rooli, joka kollektiivisesti sisältää oikeudet. Oikeuksia hallitaan vain roolien kautta [Ferraiolo et al., 2007].

Kuvassa 1 on esitetty ER-kaavio roolipohjaisen pääsynhallinnan perusperiaatteesta. Useampi kuin yksi käyttäjä voi luonnollisesti kuulua yhteen rooliin ja käyttäjä voi kuulua moneen rooliin. Roolilla voi olla useita oikeuksia ja sama pääsyoikeus voi olla monella roolilla.

Kuva 1. ER-kaavio roolipohjaisesta auktorisoinnista.

Ferraiolon ja muiden [2007] mukaan roolipohjaisen pääsynhallinnan etu on, että sitä on helppoa hallita, koska oikeuksia tarvitsee muuttaa vain rooleille, eikä monia yksittäisten käyttäjien tietoja tarvitse päivittää. Kun roolin oikeudet päivittyvät, kaikkien rooliin kuuluvien käyttäjien oikeudet päivittyvät automaattisesti. Toisaalta joustavuus on vähäisempää kuin pääsynhallintalistassa, koska kaikki roolin käyttäjät sidotaan samoihin oikeuksiin eikä yksittäiselle käyttäjälle räätälöidä oikeuksia erikseen.

Toisena etuna roolit vastaavat oikean elämän rooleja ja tehtäviä. Organisaation tietojärjestelmän rooleiksi voidaan käyttää jo organisaatiossa olemassa olevia rooleja. Yrityksen omassa tietojärjestelmässä voi olla vaikkapa työntekijän, projektipäällikön, myyntiosaston esimiehen ja toimitusjohtajan roolit. Käyttöjärjestelmissä on yleensä pääkäyttäjä (*admin* tai *root*), jolla on enemmän oikeuksia kuin tavallisella käyttäjällä ja sitä käytetään usein ja se ymmärretään auktorisoinnissa roolina. Roolit ovat siten ymmärrettävä termi pääsynhallinnan ulkopuolella, minkä takia roolipohjainen pääsynhallinta on helppo omaksua, mikä parantaa käytettävyyttä.

Jos sovelluksen roolit on määritelty huonosti, ne eivät sovi kyseisen ohjelman tai tietojärjestelmän käyttöön, jolloin roolit ovat tehottomia. Organisaation roolit eivät välttämättä sovi suoraan ohjelman pääsynhallintaan. Jos rooliksi määritellään työntekijä ja osa työntekijöistä saa tulostaa ja osa ei, työntekijäroolin tasolla ei voida määrittää onko työntekijöillä lupa tulostaa. Tällöin tulostamiseen tulisi olla erillinen rooli.

Roolien rakennus (*role engineering*) tarkoittaa ohjelmiston roolien ja roolien oikeuksien määrittelyä ennen kuin ohjelmistoa aletaan toteuttamaan. Roolien rakennukseen käytetään joko dataa käyttäjien luvista ja toiminnoista ohjelmassa tai vaatimusmäärittelyn käyttötapauksia, joiden perusteella roolit muodostetaan [Colantonio et al., 2012]. Tavoitteena on jakaa luvat rooleille siten, että roolien ja oikeuksien määrä on tasapainossa, jolloin rooleja tai oikeuksia ei ole liikaa ohjelman tarkoitukseen nähden. Lisäksi halutaan

parantaa tietoturvaa estämällä oikeuksien leviäminen niille, joilla ei saisi olla oikeutta [Coyne and Davis, 2008].

Roolit voivat muodostaa hierarkioita siten, että rooli perii toisen oikeudet [Ferraiolo et al., 2007]. Kun alempi rooli B perii ylemmän roolin A, B saa käyttöönsä A:n oikeudet. Roolille B voidaan lisätä omia oikeuksia, joita A:lla ei ole, jolloin B laajentaa A:ta. Perintä toimii melko samalla tavalla kuin olio-ohjelmoinnissa, missä aliluokka perii yliluokan piirteet eli attribuutit ja metodit lisäten uusia piirteitä. Matematiikan joukko-opin termeillä A on B:n osajoukko.

Colantonion ja muiden [2012] mielestä roolihierarkiat voidaan korvata kokonaan hierarkiattomalla rakenteella, jossa periytyväksi tarkoitetut ominaisuudet on kopioitu toiselle roolille. Organisaation luvat eivät periydy toisilta, eikä selkeitä perimyssuhteita välttämättä ole. Esimerkiksi blogissa blogin kirjoittaja, joka saa lukea ja editoida blogiaan, ei peri blogin lukemisoikeutta tavalliselta käyttäjältä, vaan on helpompi ajatella, että molemmat vain saavat lukea blogia.

Roolien tulisi olla tarpeeksi suuria ja eheitä kokonaisuuksia, jotta niitä olisi helppo hallita. Jos tarvitaan paljon yksikäsitteisiä rooleja, joita luodaan jakamalla vanhempirooleja useisiin lapsirooleihin, roolien määrä kasvaa helposti hallitsemattomaksi.

Esimerkkinä roolipohjaisesta pääsynhallinnasta keskustelufoorumilla voisi olla kolme käyttäjäryhmää: *ylläpitäjä, moderaattori* ja *tavallinen kirjoittaja*. Foorumin politiikkana voisi olla, että uuden viestin saavat kirjoittaa kaikki käyttäjät, viestejä saisivat poistaa ylläpitäjä sekä moderaattori ja vain ylläpitäjä voisi muokata foorumia.

Tehtävien hajauttaminen toteutuu roolipohjaisessa pääsynhallinnassa, kun roolin sisältämät oikeudet ovat useammalla kuin yhdellä käyttäjällä [Kuhn et al., 2010]. Tehtävien hajauttaminen pitää sisällään myös, ettei sama käyttäjä ole monessa roolissa kerrallaan. Tehtävien hajauttaminen voidaan jakaa kahteen eri tyyppiin, dynaamiseen ja staattiseen. Niiden erona on aika: dynaamisessa käyttäjä ei voi olla monessa roolissa yhtä aikaa ja staattisessa käyttäjä ei voi olla monessa roolissa ylipäätään [Ferraiolo and Kuhn, 1992].

2.6 Attribuuttipohjainen auktorisointi

Attribuuttipohjaisessa pääsynhallinnassa subjekteilla tai objekteilla on attribuutteja, joiden mukaan oikeudet määräytyvät. Attribuutti on ominaisuus, joka luonnehtii subjektia tai objektia. Säännöt määritellään attribuuttien yhdistelmien perusteella, jolloin tietyllä attribuutilla tai attribuuttiyhdistelmällä pääsy sallitaan tai evätään. Jotkut attribuutit ovat pysyviä ja ne vaikuttavat koko objektiin, kuten tiedoston tekijän nimi. Toiset attribuutit vaikuttavat vain osaan objektista ja ovat muuttuvia, kuten tiedoston muokkaajan nimi [Hu et al., 2014].

Attribuuttipohjaisessa auktorisoinnissa otetaan huomioon myös tilannetekijät (*environment conditions*), jotka muuttavat oikeuksia. Tilannetekijät ovat osa kontekstia sijaiten subjektien ja objektien ulkopuolella. Esimerkkeinä tilannetekijöistä mainitaan paikka

ja aika. Oikeus voidaan rajata esimerkiksi työajan mukaan, jolloin subjekti pääsee käyttämään järjestelmää vain työajalla. Järjestelmä päättää käyttäjän pääsystä objektiin attribuuttien ja tilannetekijöiden mukaan [Hu et al., 2014].

Attribuuttipohjainen pääsynhallinta sopii tilanteisiin, jossa oikeuksia on paljon ja ne vaihtelevat paljon eri käyttäjien välillä ja sopiva roolitus on vaikeaa. Tässä tapauksessa eri roolien määräksi tulisi helposti enemmän kuin on mahdollista hallita (*role explosion*). Toisin kuin roolipohjaisessa pääsynhallinnassa, jossa subjekteilla on roolinsa mukana tulevat tietyt oikeudet, attribuuttipohjaisessa pääsynhallinnassa subjektille voi helposti antaa yksilöllisen joukon attribuutteja. Tällöin attribuutit tekevät subjektista dynaamisemman kuin rooli.

Kuhnin ja muiden [2010] mukaan attribuuttipohjainen pääsynhallinta on helpompi toteuttaa kuin roolipohjainen, koska rooleja ei tarvitse louhia ja roolitusta rajata. Toisaalta oikeuksien vaihtaminen ja analysoiminen on vaikeampaa, koska eri oikeusyhdistelmiä voi olla 2^n, jossa n on attribuuttien määrä.

Attribuutit voidaan lisätä roolipohjaiseen pääsynhallintaan. Tämän kaltaisella hybridimallilla voidaan saada molempien hyvät puolet [Kuhn et al., 2010].

Attribuuttipohjaisuutta käytetään tietokannassa dynaamiseen peittämiseen (*dynamic data masking*), jossa kyselyiden vastauksia muokataan ajon aikana attribuuttien pohjalta ja osa vastausten sisältämistä tiedoista piilotetaan tai salataan. Kun käyttäjä tekee tietokantakyselyn, siitä otetaan attribuuteiksi käyttäjän rooli, kyselyn tyyppi, toimialue ja aika, joiden pohjalta tarkastetaan pääsyoikeudet. Tietokannassa voi olla käytäntönä, etteivät muut kuin ylläpitäjä saa nähdä SELECT-kyselyllä id-sarakkeita tai tietyiltä toimialueilta ei pääse tekemään kyselyitä yöllä välillä 0:00-7:00.

2.7 Relaatiopohjainen pääsynhallinta (RelBAC)

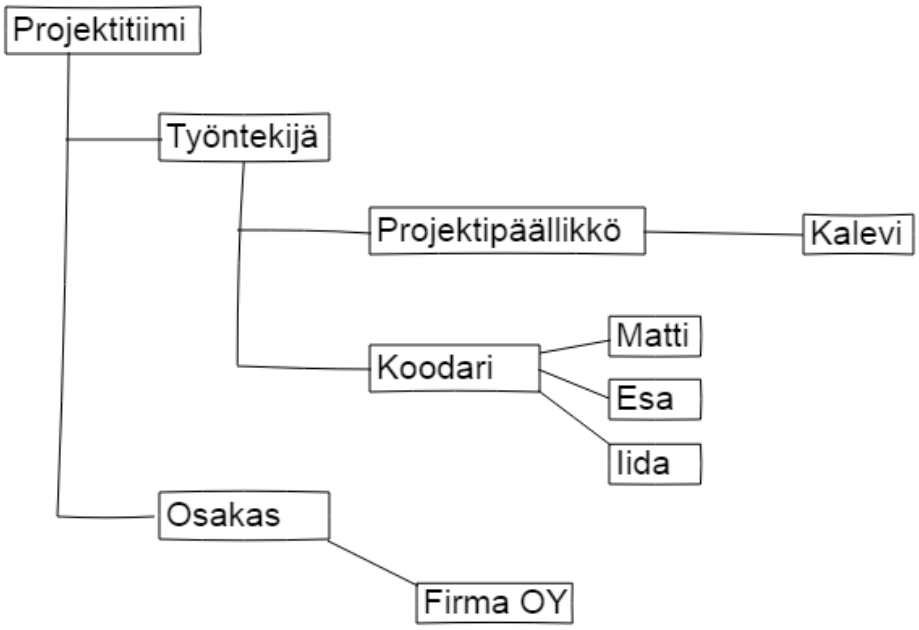
Relaatiopohjainen pääsynhallinta perustuu relaatiotietokantaan ja ER-malliin. Subjekti ja objekti ovat entiteettejä ja oikeus relaatio niiden välillä [Zhang, 2010]. Relaatiopohjainen auktorisointi määrittelee vain sen sisäisen tietorakenteen, jossa pääsynhallintaan on yhdistetty tietokantojen relaatiot, mutta sen toteutus voi muistuttaa roolipohjaista tai attribuuttipohjaista pääsynhallintaa.

Tietokannoissa entiteetti on tiedoksi mallinnettu asia ja relaatio asioiden välinen suhde. Relaatiot voivat olla yhden suhde yhteen, yhden suhde moneen tai monen suhde moneen -tyyppisiä. Relaatioista voidaan johtaa tietokantatauluja, jossa on sarakkeina entiteetin attribuutit ja riveinä yksittäiset alkiot [Harrington, 2009].

Relaatioista näkee millä subjekteilla on oikeus objektiin ja mihin objekteihin yhdellä subjektilla on oikeus. Relaatiopohjainen pääsynhallinta toteuttaa oikeuksien tarkastelun kahteen suuntaan, toisin kuin pääsyhallintalista. Zhangin [2010] mukaan relaation voi-

daan ajatella muodostavan parin *Lupa*(*subjekti*,*objekti*), joka vastaa yhtä pääsynhallinta-matriisin alkiota, mutta relaatiossa ei ole tyhjiä alkioita, jotka vievät turhaa tilaa ja ovat matriisin haittapuoli.

Kaikki subjektit, oikeudet ja objektit sisältävä iso tietokantataulu, jossa jokainen rivi sisältää yhden subjektin oikeuden yhteen objektiin, on sellainen pääsynhallintalistan ja oikeuslistan yhdistelmä, joka ei suosi kumpaakaan suuntaa. Taulu voidaan lajitella joko subjektien tai objektien mukaan, jolloin nähdään joko subjektin kaikki oikeudet objekteihin tai objektin kaikki subjektit [Sandhu and Samarati, 1994].



Kuva 2. Subjektien roolihierarkia.

Kuten roolipohjaisessa, relaatiopohjaisessa pääsynhallinnassa voi olla hierarkioita, koska taulun rivillä eli yhdellä subjektilla, objektilla tai oikeudella voi olla 1..n viittaus omaan tauluunsa, jolloin rivillä on useampi lapsi [Zhang, 2010]. Hierarkiat voidaan esittää puurakenteena, jossa juurisolmu on hierarkian ylin entiteetti ja solmujen lapset ovat aina hierarkian alempia solmuja. Kuva 2 on esimerkki projektitiimin roolihierarkiasta, jossa hierarkian ylin rooli on projektitiimi, jolla on alarooleina työntekijä ja osakas. Työntekijä-roolilla on alarooleina projektipäällikkö ja koodari. Relaatiopohjaisuutta on luonnollista käyttää tietokannoissa, jotka sisältävät jo relaatiotauluja ja pääsynhallinnalle voidaan lisätä pari taulua.

2.8 Yhteenveto

Auktorisoinnin tyypeistä keskeisimmät ovat pääsynhallintalista sekä roolipohjainen ja attribuuttipohjainen pääsynhallinta, koska ne ovat yleisimmin käytössä. Relaatiopohjainen pääsynhallinta on lähinnä tieteellisen tutkimuksen kohde eikä sitä juurikaan käytetä

sovelluksissa eikä mikään web-sovelluskehys sitä toteuta. Seuraavassa luvussa määritellään sovelluskehykset yleisesti ennen kuin esitellään tarkemmin web-sovelluskehykset.

3 Sovelluskehykset

Sovelluskehys (*Software Framework*) toimii runkona, jonka päälle sovellukset rakennetaan. Sovelluskehykset tarjoavat valmiita osasia, jotka toteuttavat ainakin sovelluksen perustoiminnallisuuden ja jota ohjelmistokehittäjät voivat käyttää ja laajentaa projektin tarpeisiin. Ne voivat sisältää ohjelmointiympäristön (*IDE*) eli tekstieditorin, jolla voi kirjoittaa lähdekoodin sekä kääntää, ajaa ja testata ohjelman [Butterfield and Ngondi, 2016].

Fayadin ja Schmidtin [1997] mukaan sovelluskehysten ominaisuuksia ovat modulaarisuus (*modularity*), uudelleenkäytettävyys (*reusability*), laajennettavuus (*extendibility*) ja kontrollin kääntö (*inversion of control*). Modulaarisuus tarkoittaa, että kehyksen ominaisuudet on jaettu kirjastoihin (*library*) eli moduleihin, ja yhden kirjaston voi vain lisätä käyttöön ja joka toimii sellaisenaan. Modulaarisuuteen kuulu myös, että kirjaston ominaisuudet on kapseloitu eli eristetty muusta koodista ja sitä käytetään ohjelmissa rajapinnan kautta, jolloin moduliin tehdyt muutokset vaikuttavat mahdollisimman vähän ulospäin. Uudelleenkäytettävyydellä sovelluskehykset tarjoavat valmiiksi geneerisiä usein käytettyjä ominaisuuksia, joita voidaan käyttää uudelleen eri toteutuksissa, jolloin tuottavuus paranee, kun asioita ei tarvitse ohjelmoida uudestaan. Laajennettavuudella kehykseen voidaan tehdä uusia ominaisuuksia, jotka toteuttavat uusia toimintatapoja sekä innovaatioita ja jotka eivät vaikuta alkuperäisiin ominaisuuksiin.

Kontrollin kääntö tarkoittaa vastuun ohjelman toiminnasta olevan ohjelmistokehyksellä. Normaalisti kontrolli ohjelman hallinnasta on ohjelmalla itsellään, mutta sovelluskehysten valmiita ominaisuuksia käytettäessä niiden oikeanlainen toiminta on sovelluskehyksen tekijöiden vastuulla. Ohjelmoija laajentaa kehyksen ominaisuuksia ohjelmansa käyttöön ja kehys kutsuu niitä koodissa, kun niitä tarvitaan; tämä erottaa sovelluskehyksen kirjastosta. Kirjasto taas tarjoaa valmiita esiohjelmoituja metodeita ja ominaisuuksia, joita kutsutaan ohjelmasta [Fowler, 2015].

Sovelluskehykset on kohdistettu tietyntyyppisille sovelluksille ja alustoille, kuten mobiilisovelluksille, työpöytäsovelluksille, web-sivuille, koneoppimiseen ja tiedonhallintaan. Eri käyttöjärjestelmille on omat sovelluskehyksensä: työpöytäsovelluksille Windowsilla .Net, macOs:lla Cocoa ja monelle käyttöjärjestelmälle kääntyvä Qt. Mobiilisovelluksille monialustainen Xamarin, osX:llä Cocoa Touch ja Androidilla Android SDK, josta lisää luvussa 7. Sovelluskehys on kirjoitettu jollain ohjelmointikielellä, kuten Spark Javalla, Node.js JavaScriptilla, Luminus Clojurella ja Django Pythonilla.

Osa kehyksistä tarjoaa tuen usealle ohjelmointikielelle, kuten koneoppimiseen tarkoitettu kehys Apache MXNet, joka toimii kielillä Python, Scala, R, Julia, C++ ja Perl. Tällöin Apache MXNet:ssä on jokaiselle kielelle oma API, joka on toteutettu kyseisen kielen syntaksilla, mutta kehyksessä taustalla toimiva laskenta on joka tapauksessa samaa. API (*Application Programming Interface*) tarkoittaa rajapintaa, jolla käytetään kehyksen ominaisuuksia ohjelmassa funktioiden ja proseduurien kautta [Butterfield and Ngondi, 2016].

Ohjelma kommunikoi kehyksen toteuttamien toiminnallisuuden kanssa API:n kautta, jonka avulla kehys voidaan integroida ohjelmaan mukaan.

Lisäksi sovelluskehyksiä on eri laajuisiin sovelluksiin. Sovelluskehyksillä voidaan tuottaa laajoja yritystason sovelluksia sekä pienempiä systeemitason sovelluksia ja väliohjelmistosovelluksia [Fayad and Schmidt, 1997]. Osa kehyksistä on tarkoitettu pieniin ja osa suurempiin sovelluksiin. Web-sovelluskehyksistä isot *full-stack* -kehykset toteuttavat paljon ominaisuuksia sisältäen tuhansia rivejä koodia ja ovat tarkoitettuja monimutkaisille web-sivuille. Vain välttämättömimmät ominaisuudet sisältävät ns. mikrokehykset on tarkoitettu yksinkertaisempien web-sivujen tekemiseen. Seuraavassa luvussa käsitellään tarkemmin web-sovelluskehyksiä.

4 Web-sivut ja web-sovelluskehykset

Jotta voitaisiin ymmärtää web-sovelluskehysten termejä, käydään ensin läpi web-sivujen ja HTTP:n toimintaa yleisesti.

4.1 HTTP ja web-sivut

Asiakas-serveri -arkkitehtuurissa (*Client-server architecture*) ryhmä asiakkaita on kytketty palvelinkoneeseen. Asiakkaaksi voidaan kutsua selainta tai laitetta, jolla käyttäjä käyttää verkkoa ja serveri toimii www-sivun säilytyspaikkana ja alustana [Butterfield and Ngondi, 2016].

URL-osoitteessa (*http://www.esimerkki.com/osa/osa2/index.html*) on alussa protokolla (*http:*), keskellä isäntä (*www.esimerkki.com*) ja lopussa polku eli kauttaviivoin eroteltuja resursseja (*/osa/osa2/index.html*). Isäntä (*host*) on url-osoitteen pääosa, jolla palvelin tunnistetaan, jossa on *hostname* (*www*) sekä verkkotunnus (*esimerkki.com*). Verkkotunnus (*domain name*) loppuu päätteeseen (*top-level domain*) (*.com*). Resurssi polun lopussa voi olla tiedosto (index.html) [Schafer, 2010; Butterfield and Ngondi, 2016].

Web perustuu HTTP-protokollaan (*Hypertext Transfer Protocol*), joka määrittelee verkon välisen viestinnän. Asiakaskone lähettää verkkosivusta serverille pyynnön (*request*), jonka otsakkeessa (*header*) on pyyntörivi, joka sisältää HTTP-metodin ja pyydetyn resurssin suhteellisen polun. Metodilla määritellään mitä pyyntö tekee, esimerkiksi yleisesti käytetty GET-metodi pelkästään noutaa sivun. Pyynnön otsakkeen seuraavalla rivillä on isännän osoite.

Serveri antaa asiakaskoneelle vastauksen (*response*), jonka otsake sisältää statusrivin, joka määrittelee, onnistuiko pyyntö. Statusrivejä ovat esimerkiksi ”200 ok”, ”403 unauthorized” ja ”404 not found”. Sekä pyynnön että vastauksen otsakkeissa voi olla lisätietoina parametririvejä. Vastauksessa on yleensä *body* eli varsinainen www-sivun sisältö. Pyynnössä on *body* silloin kun asiakaskoneelta lähetetään dataa POST tai PUT -metodia käyttäen [Fielding et al., 1999; Belshe et al., 2015].

HTML (*hypertext-markup-language*) on merkkauskieli web-sivujen rakenteen esittämiseen, jossa eri elementit esitetään tageilla. CSS (*cascading style sheets*) on kieli web-sivujen ulkoasun määrittelemiseen. Eväste (*cookie*) on pieni tieto käyttäjän toimista ja tilasta web-sivulla, jonka web-sivu asettaa käyttäjän kovalevylle [Butterfield and Ngondi, 2016].

Web-sivu on hypertekstiä sisältävä dokumentti internetissä [Butterfield and Ngondi, 2016]. Web-sovellus on internetin kautta toimiva sovellus, jonka sisältö on dynaamista eli se muuttuu tilanteen mukaan ja käyttäjien toimesta. Termejä käytetään myös toistensa synonyyminä, mutta yleensä web-sovellus on web-sivua monimutkaisempi ja isoja sovelluksia on mahdoton toteuttaa pelkillä HTML-tiedostoilla, jotka on tarkoitettu yksinkertaisemmille staattisille web-sivuille. Monimutkaisempiin web-sivuihin tarvitaan web-sovelluskehyksiä.

4.2 Web-sovelluskehykset

Web-sovelluskehys (*Web application framework*) eli web-kehys on verkkoympäristössä toimiva ohjelmistokehys, jolla tehdään web-sovelluksia. Samoin kuin sovelluskehyksiin yleisesti, niihin on esiohjelmoitu paljon yleisesti tarvittavaa toiminnallisuutta, joita tarvitaan web-sivuilla ja joita ohjelmoijan ei tarvitse tehdä itse uudelleen, jolloin he voivat keskittyä itse sivun sisältöön [Butterfield and Ngondi, 2016].

Sovelluskehykset toteuttavat Makain [2017] mukaan yleensä:

- tietokannan kanssa kommunikoinnin,
- käyttäjien tunnistautumisen,
- sessiot (käsitellään tarkemmin alakohdassa 4.4.1),
- reitityksen,
- mallineet,
- pyyntöjen ja vastauksien sisällön manipuloinnin olioiden kautta ja
- tietoturvaa, kuten suojauksen CSRF-hyökkäyksiltä.

Sovelluskehyksiin voi tehdä ja ladata erillisiä kirjastoja, jotka sisältävät lisäominaisuuksia, jolloin toteutuu sovelluskehysten modulaarinen rakenne ja laajennettavuus. Kirjastot eli modulit tarjoavat ohjelmointiyhteisölle tavan lisätä toiminnallisuutta pieninä osina. Myös auktorisointi on toteutettu sovelluskehykseen joko valmiina tai erikseen ladattavana osana.

Malline (*template*) on dynaaminen yleiskäyttöinen sapluuna html-tiedostoille. Mallineiden kieli on usein eri kuin web-kehyksen kieli ja kehys muuttaa ne käyttäjälle html-muotoon. Mallineet sisältävät html-elementtejä muistuttavien rakenteiden lisäksi muuttujia sekä kontrollirakenteita, kuten if-lausekkeita, joilla voidaan muuttaa sivun ulkoasua dynaamisesti.

Reititys (*routing*) tarkoittaa mekanismia, jolla web-kehys palauttaa tietyllä pyynnöllä tietyn sivun. Sama url-osoite voidaan asettaa palauttamaan eri HTTP-metodeilla eri sivu. Teknisesti yksi reitti on jonkinlainen määritys, jossa yksi tai useampi url-osoite asetetaan osoittamaan tiettyyn kohtaan koodissa, joka lopulta palauttaa tietyn sivun. Yksi reitti määritellään usein kutsumalla jotain web-kehyksen toteuttamaa reititysmetodia url-osoitteella ja HTTP-metodin nimellä. Joissain web-sovelluskehyksissä voidaan tehdä oma reititysfunktio, jossa määritellään mitä tehdään, kun pyyntö saapuu.

Web-sovelluskehys tarjoaa rungon, jota täydennetään omalla koodilla. Kehys esimerkiksi määrittää mallineille syntaksin ja kansion, jossa ne sijaitsevat. Ohjelmoija toteuttaa mallineet kehyksen määräämin tavoin, jonka jälkeen kehys käyttää niitä palvelimella. Ohjelmoija toteuttaa reitityksen sovelluskehyksen määräämällä tavalla ja sovelluskehys käyttää reittejä ajon aikana. Näin kontrollin kääntö toteutuu web-sovelluskehyksissä.

Web-sovellukset jaetaan kahteen osaan: käyttäjille näkyvät asiat kuuluvat sovelluksen selainpuoleen (*front-end*) ja taustalla toimiva logiikka sekä datan käsittely palvelinpuoleen (*back-end*) [Butterfield and Ngondi, 2016]. Nämä kytkeytyvät asiakas-serverimalliin siten, että *front-end* toimii asiakkaan puolella ja *back-end* serverin puolella. Selainpuoleen kuuluu käyttäjän kanssa kommunikointi, sivuston käyttöliittymä sekä ulkoasu ja ulkoasun toteuttamisen työkalut eli mallineet, HTML ja CSS. Palvelinpuoleen kuuluu tulevan ja lähtevän tiedon käsittely ja varastointi sekä tietokannat. Kun käyttäjä tunnistautuu, käyttäjälle näkyvä lomakkeen täyttäminen ja ulkoasu on *front-end* -puolta ja lomakkeen tietojen käsittely *back-end* -puolta. Auktorisoinnissa päätös käyttäjän pääsystä tehdään tiedon ja ohjelmoidun logiikan perusteella palvelimen puolella ja käyttäjä näkee tai ei näe objektia selaimen puolella.

Sovelluskehystä voidaan kutsua pelkästään selainpuolen (*front-end framework*) tai palvelinpuolen (*back-end framework*) sovelluskehykseksi ja *Full-stack framework* sisältää sekä selain- että palvelinpuolen. Jaottelu näiden välillä ei kuitenkaan ole selkeää, koska useat varsinaisesti palvelinpuoleen brändätyt kehykset sisältävät myös selainpuolen asioita ja toisin päin. Pääasiassa palvelinpuolen kehys Django sisältää mallineet, jotka liittyvät sivuston ulkoasuun ja siten selainpuoleen. Reititys on perinteisesti hoidettu palvelimen puolella, jolloin palvelin on ohjelmoitu lähettämään tietty sivu tietyllä selaimelta saadulla osoitteella, mutta *front-end* -kehyksessä reititys voidaan hoitaa kokonaan selaimessa siten, että kehys näyttää sivusta tietyn osan selaimen osoitteen mukaan. Jälkimmäisessä sivua ei ladata palvelimelta uudestaan, vaan kehys jakaa sitä osiin selaimessa.

4.3 Cross-Origin Resource Sharing

Cross-Origin Resouce Sharing (CORS) on tapa antaa web-sovellukselle pääsy toiselle toimialueelle. Kahdella palvelimella on eri alkuperä (*origin*), jos niillä on eri protokolla, portti tai isäntä, jolloin ne kuuluvat eri toimialueeseen.

Oletuksena sellaiset yleiset pyynnöt toiselle toimialueelle ovat luvallisia, jotka käyttävät vain metodeita GET, HEAD tai POST ja yleisiä otsakkeita (kuten *Connection, Accept ja Width*) ja ne sisältävät yleisiä sisältötyyppejä (tekstitiedosto, html-tekstitiedosto, lomake). Muunlaiset pyynnöt ovat toimialueiden välillä oletuksena luvattomia, ellei niitä ole sallittu CORS-otsakkeilla. Toisesta toimialueesta olevien tagien upotus (css, script, img) on sallittua, mutta niiden tietojen suora lukeminen on kielletty. CORS:a tarvitaankin, jos toiselta toimialueelta halutaankin lukea jotain suoraan datana, esimerkiksi JSON-muodossa. [Mozilla Foundation 2, 2017; Mozilla Foundation 4, 2017]

CORS toimii asettamalla otsakkeita pyyntöön. Muissa kuin oletuksena sallituissa pyynnöissä lähetetään ennen varsinaista pyyntöä OPTIONS-metodia käyttävä tarkastuspyyntö (*preflight*), johon salliva palvelin lähettää vastauksena CORS-otsakkeet. Tarkastuspyynnöllä varmistetaan varsinaisen pyynnön olevan sallittu, joka lähetetään vasta tarkastuspyynnön jälkeen.

CORS-otsakkeita ovat:

- *Access-Control-Allow-Origin*, sallitut toimialueet, *-merkillä kaikki toimialueet,
- *Access-Control-Allow-Methods*, varsinaisessa pyynnössä sallitut HTML-metodit,
- *Access-Control-Allow-Headers*, varsinaisessa pyynnössä sallitut otsakkeet ja
- *Access-Control-Max-Age*, tarkistuspyynnön säilytysaika sessiossa.

4.4 Autentikointi

Autentikointi tarkoittaa subjektin eli yleensä käyttäjän tunnistautumista [Butterfield and Ngondi, 2016]. Autentikoinnin tehtävänä on varmistaa, että subjekti on se, kuka väittääkin olevansa. Tunnistautuminen liittyy läheisesti auktorisointiin, koska käyttäjän tulee olla tunnistettu ennen kuin käyttäjän pääsy voidaan evätä tai sallia. Pääsynhallinnassa tulee myös tilanteita, joissa vain sisäänkirjautuneiden käyttäjien pääsy sallitaan tiettyihin web-sovelluksen osiin. Usein autentikoinnista kirjoitetaan samassa yhteydessä pääsynhallinnan kanssa, joten sitä käsitellään tässäkin pitkästi.

Perinteisesti käyttäjä tunnistautuu käyttäjänimellä ja salasanalla, jotka palvelin varmistaa oikeiksi. Subjekti voi tunnistautua myös muilla tavoin, esimerkiksi kolmannen osapuolen palvelun avulla. OAuth2:ssa sovellus tunnistetaan sovellukselle annetuilla *ClientID* -tunnuksella ja *ClientSecret* -salasanalla [Parecki, 2018]. Verkkopankeissa käytetään avainlukulistan numeroa, jonka tietää vain asiakas.

4.4.1 Sessiot

Tunnistautumisen jälkeen palvelin luo istunnon eli session. Sessiolla on jonkinlainen tunnus, jota kutsutaan myös nimellä session ID tai avain. Session tunnusta käytetään sekä itse session että käyttäjän yksilöimiseen, siitä pidetään kirjaa sovelluksessa ja se lisätään myös käyttäjän puolelle evästeeseen. Koska sessio on käynnissä tai loppunut, palvelimella ja käyttäjällä on jonkinlainen yhteinen tila.

Tietoturvan vuoksi session tunnuksen tulisi olla pitkä ja muodostua satunnaisesti numeroista ja kirjaimista, jolloin eri tunnuksia voi olla monia ja session tunnusta on vaikea arvata [Nikiforakis et al., 2011]. Tunnuksien generoiminen ja asettaminen on toteutettu web-kehyksissä valmiiksi. Käyttäjänimi ja salasana lähetetään käyttäjältä serverille vain kerran, mikä vähentää niiden vuotamisen riskiä, muilla kerroilla käytetään session tunnusta.

Käyttäjän selaimeen asetettu eväste olisi hyvä asettaa *HttpOnly*-tyyppiseksi, jolloin sitä ei saa näkyviin käyttäjän puolella skriptillä (esim. *document.cookies*). Toiseksi evästeen olisi hyvä olla allekirjoitettu (*signed*), jolla serveri varmistaa, ettei käyttäjä ole muokannut sitä [Nemeth, 2015]. Evästeelle tulisi antaa lyhyt voimassaoloaika, mikä vähentää aikahaarukkaa, jossa session tunnus voidaan kaapata. Tunnus poistuu evästeestä, kun käyttäjä kirjautuu ulos tai eväste vanhenee.

4.4.2 HTTP Basic Authentication ja Digest Access

HTTP Basic Authentication -metodissa ei käytetä evästeitä eikä sessioita, vaan joka käyttäjältä saatu pyyntö sisältää tunnistautumistiedot [Nemeth, 2015]. Käyttäjänimi ja salasana on yhdistetty merkkijonoksi, jossa ne on eroteltu kaksoispisteellä ja sen jälkeen Base64-enkoodattu uudeksi lyhemmäksi merkkijonoksi. Pyyntöjen Authorization-otsake sisältää Basic-avainsanan ja sen jälkeen edellä mainitun Base64 -merkkijonon. Basic-sana yksinkertaisesti kertoo serverille, että autentikaatiotapa on *HTTP Basic Authentication* [Reschke, 2015].

Tapa on tietoturvan kannalta ongelmallinen, koska Base64-koodaus ei varsinaisesti salaa tunnistautumistietoja ja tiedot sisältyvät moneen pyyntöön, joista kaikista hyökkääjä voi ne noukkia ja purkaa Base64-muodon. Lisäksi metodi ei tue uloskirjautumista eikä salasanan vaihtoa [Nemeth, 2015]. Tämän vuoksi *HTTP Basic Authentication* -metodia ei yleensä käytetä.

HTTP Digest Access -metodissa pyynnössä lähetetään tiivistemerkkijono. Tiiviste sisältää välimerkein erotellut käyttäjänimen, url-osoitteen ja salasanan MD5-algoritmilla tiivistettynä. Näiden lisäksi tiiviste voi sisältää myös *nonce*-arvon, joka on palvelimelta saatu tunniste ja vastaavasti *cnonce* on käyttäjältä saatu tunniste [Franks et al., 1999]. Sinänsä tiivisteen tekeminen on turhaa, koska moderneissa tietokannoissa salasana on jo tiivisteenä, eikä sitä ei saa tarkistettua sellaisenaan.

Sekä *HTTP Basic Authentication* että *HTTP Digest Access* -metodit ovat yksinkertaisia ja alttiita XSS-hyökkäykselle. XSS-hyökkäys (*cross site scripting*) tarkoittaa haitallisen koodin lisäämistä luotettavasta lähteestä saatuun dataan. Data voi olla selaimesta tai toiselta sivulta saatu pyyntö. XSS-hyökkäys tapahtuu selaimen ja palvelimen tai kahden palvelimen välisessä verkossa, jonka vuoksi sitä kutsutaan myös mies-välissä-hyökkäykseksi (*man-in-the-middle attack*). Injektoitu skripti muuttaa esimerkiksi pyynnön osoitteen hyökkääjän sivulle, joka kaappaa tiedot [Morgan, 2010; Butterfield and Ngondi, 2016].

4.4.3 JSON Web token

JSON Web token eli JWT-metodissa otsakkeissa lähetetään Base64-enkoodattu JSON-objekti, jossa on kolme osaa, jotka on erotettu pisteillä. Ensimmäisenä osana JSON:ssa on käytetyn hajautusalgoritmin sisältävä otsake ja toisena subjektin tiedot sisältävä tietosisältö [Jones et al., 2015]. Viimeisenä osana on signeeraus, jossa kaikki edelliset osat on Base64-enkoodattu ja salattu hajautusalgoritmilla serverin pyynnössä antaman salasanan (*secret*) kanssa. Signeeraus-osalla varmistetaan, että JWT-tokenin lähettäjä on sama, jolle salasana on aiemmin lähetetty eikä se ole muuttanut tietoja [Nemeth, 2015]. Eri salasana ja muuttunut tietosisältö muuttaisi hajautusalkion täysin erilaiseksi.

JWT-metodissa ei käytetä sessioita eikä evästeitä, jolloin tieto subjektista kulkee JWT-tokenissa, joka on tallennettu käyttäjän puolelle. Tämä sopii yhteen REST-protokollan kanssa, sillä REST on tilaton järjestelmä, jossa mitään tietoa ei tallenneta palvelimelle, vaan session tieto on asiakkaan puolella ja tieto kulkee vain viesteissä asiakkaan ja palvelimen välillä [Fielding and Taylor, 2000]. JWT-token on altis XSS-hyökkäyksille, koska se tallennetaan selaimen tietovarastoon (*local storage*) [Nemeth, 2015].

4.4.4 Kertasalasana

Tunnistautumisen jälkeen palvelin generoi algoritmilla kertasalasanan (*one-time-password*), jonka se lähettää takaisin palvelimelle. Kertasalasana on vain asiakkaan ja palvelimen tiedossa ja sen tulisi olla mahdollisimman satunnainen. Kertasalasanan generoinnin satunnaisuus perustuu muuttuvaan alku-arvoon (*seed*), johon käytetään joko nykyistä ajanhetkeä tai nousevaa laskuria. Kertasalasanaa käytetään usein muiden tunnistautumismenetelmien ohessa kaksinkertaiseen todentamiseen [Nemeth, 2015].

Kertasalasana eroaa session tunnuksesta siten, että se on nimensä mukaisesti voimassa vain yhden kerran, jonka jälkeen se vanhentuu ja sitä ei voi käyttää uudestaan. Session tunnus voi olla toteutuksesta riippuen samalla käyttäjällä sama useassa eri sessiossa, mutta kertasalasana generoidaan aina erikseen.

4.5 MVC-malli

Sovelluskehykset käyttävät suunnittelumalleja eli yleisiä vakiintuneita tapoja tehdä asioita. Suunnittelumalli on hyväksi havaittu ratkaisu johonkin ongelmaan [Gamma et al., 1995]. Yksi tällainen suunnittelumalli on MVC-malli, joka sisältää kolme komponenttia, malli (*Model*), näkymä (*View*) ja kontrolleri (*Controller*).

Malli esittää jotain asiaa ja se sisältää tiedon, miten dataa manipuloidaan. Malli on käyttöliittymästä erillinen paikka tiedon mallinnukselle ja se luo datalle rajapinnan. Malli voi toimia edelleen esimerkiksi tietokannan kanssa ja tallentaa tiedon sinne. Kontrolleri kommunikoi mallin kanssa, se prosessoi ja ottaa mallista saadun tiedon muuttujiin ja tulkitsee mallilta saadut virheviestit. Kontrollerin tehtävänä on hakea tieto mallista ja antaa se näkymälle. Näkymä näyttää saadun datan jossain muodossa käyttäjälle [Zlobin, 2013].

Tärkeää MVC-mallissa on, että komponentit ovat erillisiä kokonaisuuksia: mallin rakenne on riippumaton kontrollerin rakenteesta ja näkymän tavasta esittää asioita. Näin ne voidaan kehittää ja testata erikseen, jolloin toisessa mallin komponentissa tehty muutos ei vaikuta muihin. Toisaalta niiden toiminta riippuu toisistaan: näkymä tarvitsee mallin tietoa, jonka se saa kontrollerilta [Zlobin, 2013].

Web-kehysten erityispiirteenä kontrollerit huolehtivat myös pyyntöjen ja vastausten logiikasta. Tietynlainen pyyntö reititetään tietylle kontrollerille, joka kutsuu näkymää, joka palauttaa vastauksen. Vastauksen runko voi olla malline, html-sivu tai esimerkiksi JSON-muotoista dataa. Näkymät voi käyttää mallineita html-sivujen generoimiseen.

5 Pääsynhallinta web-sovelluskehyksissä

Tässä luvussa esitellään ensin pääsynhallinnan toimintamekanismeja web-sovelluskehyksissä sekä pääsynhallinnan suunnittelumalleja ja -paradigmoja.

Pääsyoikeuksien sijainti vaihtelee, ne voivat kuulua joko subjektiin tai objektiin. Pääsynhallinta on resurssipohjaista, jos objekti sisältää tiedon omistussuhteista ja sallituista subjekteista. Tällöin objektin mallissa tai tietokannassa on lista rooleista tai käyttäjistä, jotka voivat käyttää objektia. Objektin omistaja saa käyttää objektia, jolloin omistaja sisältyy listaan. Auktorisointi voi myös olla subjektipohjaista, jolloin käyttäjäolio sisältää roolin, attribuutit tai muuten tiedon mitä se voi tehdä. Siinä tapauksessa subjektin rooli ja attribuutit on tallennettu tekstinä subjektimalliin tai tietokantaan. Subjektikohtaisessa pääsyhallinnassa sisäänpääsy päätetään subjektista saadun tiedon perusteella ja resurssikohtaisessa käytössä on myös tietoa objektista. Pääsynhallintakirjasto voi tarjota myös oman tapansa tallentaa pääsyoikeudet olioon.

Pääsynhallinta voi joko kuulua kehyksen valmiiksi toteuttamiin toimintoihin tai olla erikseen ladattava kirjasto. Pääsynhallintakirjasto abstraktoi ja keskittää auktorisoinnin toimintaa, joten se tarjoaa auktorisoinnin valmiiksi määritellyssä ja testatussa paketissa. Kirjastoon on kapseloitu mahdollisimman paljon pääsynhallinnan sisäistä toimintalogiikkaa, jotta ohjelmoija ei tekisi virheitä toteuttaessaan niitä itse. Pääsyhallintakirjasto tuottaa tyypillisesti auktorisoinnin mekanismit sisältävän olion, jota ohjelmoija voi käyttää koodissa ohjelman tarpeisiin. Auktorisointi toteuttaa siis olioparadigmaa.

Itse ohjelmakoodissa pääsynhallinta voidaan merkitä usealla eri tavalla rakenteellisesti, eri suunnittelumalleja käyttäen. Tavallisesti oikeuden tarkastelu tapahtuu jonkun funktion sisällä koodissa, esimerkiksi if tai switch -ehtolauseessa. Pääsynhallintaoliolta saadaan metodit oikeuksien tarkasteluun ja olio säilöö mahdollisesti myös itse oikeudet.

Java-, Python, C#- kielissä, sekä Laravel-kehyksessä on koristajia (*decorator*), joka on funktion tai objektin yhteyteen asetettu objektin nimi. Auktorisointi voidaan kytkeä päälle tietyissä funktioissa asettamalla niiden yläpuolelle tai viereen koristajan. Koristajaobjekti on määritelty toisaalla ja se toteuttaa jonkun ominaisuuden. Koristaja asettaa funktion tai objektin sen määrittelemään rajapintaan, jolloin se antaa funktiolle tai objektille jonkun toiminnon [Gamma et al., 1995]. Koristaja kytkee pääsyoikeuden tarkistamisen ennen funktiota.

Pääsynhallintakirjasto voi tarjota valmiita metodeita oikeuksien testaamiseen tai niitä voi tehdä itse ja linkittää kirjaston käyttöön. Nämä metodit palauttavat yleensä totuusarvon tai esimerkiksi olion, joka voi sisältää muutakin tietoa. Metodeista on erilaisia versioita, mutta yleensä metodille annetaan subjektin olio. Metodeista voi olla resurssikohtainen versio, jolle annetaan lisäksi objektin olio.

Pääsynhallinnan tarkistukset voivat sijaita reittien yhteydessä tai vaihtoehtoisesti ne voivat olla keskitettynä omaan luokkaansa, joka on käytössä kaikkialla. Pääsyoikeuksien

tarkistaminen koodissa perustuu loppujen lopuksi yksinkertaisiin ehtolauseisiin ja totuusarvoihin, joita ohjelmoija kirjoittaa. Pääsynhallinnan logiikka voidaan panna globaaliin funktioon, jota voidaan kutsua kaikkialta. Tällaista funktiota kutsutaan apuriksi (*helper*). Sisäänpääsy voidaan myös tarkistaa väliohjelmistossa ennen reitityksiä tai itse mallissa. Jotkut pääsynhallintakirjastot ovat joustavia sen suhteen, missä oikeuksien testaaminen voi sijaita, toisissa pääsyoikeus tarkastetaan aina tietyllä tavalla.

Web-kehyksissä oikeuksia voidaan joutua tarkistamaan myös mallineiden sisällä silloin, jos niiden avulla halutaan näyttää tai piilottaa elementtejä käyttöoikeuksien mukaan. Tällöin pääsynhallinta sisältää kontrollirakenteita web-kehyksen mallinekielellä. Kontrollirakenteella voidaan esimerkiksi tarkistaa, ettei käyttäjällä ole tiettyä roolia ja estää roolille kuuluvan linkin näkyminen mallineessa.

Joissain ohjelmistokehyksissä politiikan määritys tehdään kokonaan reittien sisällä, mutta useimmissa ne tehdään keskitetysti kerran ja yhdessä paikassa. Pääsynhallintapolitiikka on teknisesti yleensä olio, joka säilöö säännöt ainakin ajon aikana ja se tarjoaa valmiin metodin oikeuksien luomiseen. Vaihtoehtoisesti politiikka voi olla määritelty esimerkiksi JSON tai XML -muodossa, josta pääsynhallintaluokka lukee oikeudet. Luokka voi lukea oikeudet myös tietokannasta. Tapa oikeuksien luomiseen ja säilömiseen riippuu paljolti ohjelmointikielen ja kehyksen syntaksista.

Ohjelmointiteknisesti roolien nimet ja attribuutit ovat merkkijonoja, jotka ovat olioissa tai tietokannoissa. Pääsyoikeudet ovat usein totuusarvoja tai merkkijonoja. Käyttäjäoliossa tietty merkkijono merkitsee tiettyä roolia. Ne ovat yksikäsitteisiä, joten sallittujen roolien vertailu käyttäjän rooliin onnistuu. Vastaavasti käyttäjäolion attribuutin merkkijonoa voidaan vertailla sisäänpääsyn yhteydessä.

Seuraavaksi käsitellään yksittäisiä ohjelmointikehyksiä tarkemmin, joista käydään lyhyesti läpi ensin ohjelmointikieltä ja sitä, miten auktorisointi tarkalleen toimii ohjelmistokehyksessä. Kaikkia olemassa olevia kehyksiä ja kieliä ei voi tilan- ja ajanpuutteen vuoksi esitellä tässä, joten esittelyyn on valittu vain yleisiä ja suosittuja web-sovelluskehyksiä. Käsittelen kehyksiä Express, React, Angular, Django, Spring, Asp.Net, Ruby on Rails sekä Laravel, jotka löytyvät useiden web-kehysten suosiota mittaavien listojen kärkipäästä [Hot Frameworks, 2018; Schlosser, 2018]. Valitsin myös itselleni tuttuja kehyksiä, se muodostui toiseksi valintakriteeriksi kehyksen yleisyyden lisäksi. Yhdestä ohjelmointikielestä on valittu keskimäärin yksi web-sovelluskehys, vaikka esimerkiksi PHP-kielellä on tehty paljon erilaisia kehyksiä.

Ohjelmointikielestä kerrotaan lähinnä historiaa ja ohjelmointikielen paradigma sekä tyypitys. Kappaleissa on keskitytty sovelluskehyksen esittelyyn, mutta seuraavassa kappaleessa esitellään myös Node.js-ajoympäristöä, jonka päällä melkein kaikki JavaScript-kehykset toimivat.

5.1 JavaScript, Node.js

JavaScript on kevyt, pääasiassa web-selaimissa käytetty ohjelmointikieli. Tyypitys on dynaamista eli JavaScriptin muuttujien tyyppiä ei ole määritelty käännöksen aikana, vaan tyypit asetetaan vasta ajon aikana. JavaScript on usean paradigman kieli: sillä voi ohjelmoida proseduraalisesti, funktionaalisesti ja useimmiten sitä käytetään oliopohjaisesti.

JavaScriptin oliot ovat prototyyppisiä, joita luokkatyyppisistä kielistä poiketen ei ole jaoteltu luokkamäärityksiin ja instansseihin, vaan prototyyppinen olio on itsessään sekä sen ilmentymä että abstrakti määritelmä. Jokaisella oliolla on prototyyppiolio, josta se perii ominaisuutensa, jonka saa näkyviin olion prototype-attribuutilla. Prototyyppi omistaa olion rakentimen, attribuutit ja metodit. Oliot muodostavat perintäketjuja kun olio perii jonkun prototyypin, ketjujen päässä on aina JavaScriptin pääolioluokka *Object*, josta objektit perivät esimerkiksi valueOf()-metodin. [Mozilla Foundation 1, 2017]

Myös funktiot ovat olioita, jonka vuoksi funktiolle voi antaa parametriksi toisen funktion, vastakutsufunktion (*callback function*). Ulkoisen funktion on tarkoitus kutsua parametriksi saatua sisäistä vastakutsufunktiota myöhemmin. [Mozilla Foundation 1, 2017]

JavaScriptin kehitti alun perin Netscape Communications 1995, sittemmin ECMA standardoi siitä kaikille selaimille yhteisen ja nykyään sen kehityksestä vastaa Mozilla Foundation. JavaScriptia on käytetty perinteisesti selaimen puolella, mutta Node.js:n ja muiden sovelluskehysten myötä sitä on alettu käyttää myös palvelimen puolella. JavaScriptin osaamista voi hyödyntää molemmilla puolilla ja ohjelmointikieltä ei tarvitse vaihtaa niiden välillä siirryttäessä. [Zakas, 2009; Cantelon et al., 2014]

Node.js (www.nodejs.org/en/) eli Node on JavaScript-ajoympäristö. Se käyttää Google Chromen V8-virtuaalikonetta, joka kääntää JavaScriptin suoraan natiiviksi konekieleksi käyttämättä välissä tavukoodia kuten esimerkiksi Javan virtuaalikone. V8 käyttää suoraan tehokkaampaa konekieltä tavukoodin tulkkaamisen sijasta, mikä parantaa suorituskykyä [Cantelon et al., 2014; Venners, 1998].

Node tarjoaa asynkronisen arkkitehtuurin internetin siirrännälle eli syötteelle ja tulosteelle (*Input/Output*) [Cantelon et al., 2014]. Asynkroninen tarkoittaa, että syötteen lukemisen tai tulosteen lähettämisen aikana ei pysähdytä odottamaan sen valmistumista, vaan ohjelma voi tehdä näiden tapahtumien aikana muita, muualla lähdekoodissa määriteltyjä, asioita. Asynkronisuus voitaisiin toteuttaa myös itse säikeillä (*threads*), joista jotkut säikeet odottavat siirrännän valmistumista ja muut säikeet tekevät muita asioita, mikä on monimutkaista ja virhealtista [Herron, 2013].

Datan saamiseen tai HTTP-pyyntöjen lähtemiseen saattaa kulua palvelimelta aikaa: latenssia voi aiheuttaa tietokanta- tai verkkoyhteyden ongelmat tai kiintolevyn hitaus. Node.js:ssa pyörii tapahtumasykli (*event loop*), jonka tehtävänä on tarkkailla milloin siirräntätapahtumat ovat valmiita eli syötteet lähteneet ja tulosteet tulleet. Tapahtumalle teh-

dylle funktiolle asetetaan parametriksi vastakutsufunktio, joka käsittelee tapahtuman tuloksen ja jolle tapahtumasykli lähettää tuloksen vasta, kun tapahtuma on valmis [Cantelon et al., 2014].

```
//Uses express-framework
const express = require('express');
const app = express();

app.get('/', function (request,response){
    response.sendFile(__dirname + "\\index.html");
});
```

Koodiesimerkki 1. Get-reititysmetodi.

Väliohjelmisto (*middleware*) toimii nimensä mukaisesti kahden asian välissä, perinteisesti joko käyttöjärjestelmän ja ohjelmiston tai vaihtoehtoisesti asiakkaan ja serverin välissä. Nodessa useimmat funktiot ovat väliohjelmistoa, koska pyyntöobjektit selaimelta palvelimelle ja vastausobjektit palvelimelta selaimelle kulkevat niiden kautta ja ne käsittelevät niitä välissä. Väliohjelmistofunktiot muodostavat usein pinon, jolloin ne antavat tiedot usein ketjussa eteenpäin aina seuraavalle väliohjelmistofunktiolle, joka käsittelee tietoja myöhemmin eri tavalla. Koodiesimerkissä 1 on esimerkki väliohjelmistofunktiosta Get-metodille. Väliohjelmistofunktiota kutsutaan sen jälkeen, kun pyyntö tulee käyttäjältä ja ennen kuin pyyntö menee reititysfunktioille. [Ince, 2013; Cantelon et al., 2014]

Node.js perustuu pitkälti modulaariseen rakenteeseen, jonka filosofiana on pilkkoa toimintoja pieniin palasiin. Siinä missä useiden Python- ja PHP-kehysten perustoiminnallisuus on yhtenäinen paketti, myös Noden ydintoiminnot on kokonaan jaettu pieniin moduleihin. Uusia moduleita julkaistaan *npmjs.com*-sivustolla, josta ne ladataan ja lisätään ohjelman käyttöön pakettienhallintatyökalun avulla. Moduli koostuu yhdestä tiedostosta tai kansiollisesta tiedostoja ja niistä näkyy ulospäin luokkia sekä väliohjelmistofunktioita, joita voi käyttää omassa koodissa tyypillisesti asynkronisesti [Cantelon et al., 2014].

5.1.1 Express

Express (www.expressjs.com/) on palvelinpuolen kehys Node-ohjelmointiympäristöön. Se sisältää vain välttämättömimmät ominaisuudet, toisin kuin useat muut monoliittisemmat kehykset. Express sisältää lähinnä reitityksen, serverin ylläpidon, tiedonsiirron sekä oliot pyynnöille ja vastauksille. Toimintoja ei ole sidottu tiettyyn rakenteeseen, esimerkiksi reitityksen ei tarvitse sijaita tietyssä tiedostossa [Cantelon et al., 2014].

Koska Express itsessään sisältää vähän ominaisuuksia, ohjelmoijat voivat laajentaa palettia moduleilla ja käyttää vain niitä toimintoja, joita he tarvitsevat. Moduleita hallitaan Node.js:n mukana asentuvan ja komentorivillä toimivan pakettienhallintatyökalun

npm (*Node package manager*) avulla ja myös vaihtoehtoisia pakettienhallintatyökaluja on olemassa, kuten Yarn. Yksi moduli ensin asennetaan pakettienhallintatyökalulla joko yhteen projektiin tai kaikkialle ja sen jälkeen se otetaan käyttöön muuttujaan tai muuttujiin kooditiedoston alkuriveillä joko *require* tai *import* -lauseella.

Yleisesti käytetty moduli *express-session* tarjoaa väliohjelmistofunktion, jota kutsutaan ennen reitityksiä ja joka hoitaa session tunnuksen asettamisen evästeeseen ja session datan tallentamisen palvelimelle. Toinen moduli *body-parser* jäsentää POST-pyynnöstä saadun datan pyyntöobjektin body-olioon ja mahdollistaa pyynnön käytön. *Passport*-moduli toteuttaa autentikaation, joka voi olla joka paikallinen (*local authentication strategy*) eli toimia oman sivuston kautta tai tapahtua kolmannen osapuolen, kuten Googlen tai Facebookin kautta. Käsittelen kolmea Express-modulia: Node-abac, Abac ja Accesscontrol.

Node-abac (www.npmjs.com/package/node-abac) on tarkoitettu attribuuttipohjaiseen pääsynhallintaan. Sen politiikka määritellään joko Yaml tai JSON -muodossa, joka sisältää attribuutti- ja sääntöobjektit (*rules*). Attribuutti-objektiin merkitään kaikki mahdolliset attribuutit jokaiselle subjektityypille erikseen. Säännöt-objektiin määritellään kaikki oikeudet ja jokaiseen oikeuteen tarvittavat attribuutit, jotka eri subjekteilla tulee olla, jotta käyttäjä läpäisee oikeuden tarkistuksen. Modulin pienenä erikoisuutena jokaiselle attribuutille määritellään eksplisiittisesti odotusarvo ja vertailuoperaattori usean attribuutin vertailuun, tyypillisesti odotettu arvo on totuusarvo *true* ja vertailuoperaattori && eli ”ja” [Node-Abac, 2015].

Pääsynhallinnan käyttämiseen Node-abac -kirjastossa on kaksi funktiota, *getRuleAttributes*, jolla saa sääntöön (oikeuteen) tarvittavat attribuutit ja *enforce,* jolla oikeus voidaan evätä tai sallia. *Enforce*-funktiolle annetaan parametreina subjekti, oikeuden nimi ja valinnaisena parametrina objekti, joiden perusteella se tarkistaa onko subjektilla oikeus ja palauttaa totuusarvon. Sille voi antaa kolmanneksi parametriksi objektin, jolloin se tarkistaa oikeuden objektikohtaisesti [Node-Abac, 2015].

Abac-modulissa (www.npmjs.com/package/abac) tallennustapa (”*backend*”) määrittää miten säännöt tallennetaan. Modulista löytyy valmiiksi *InMemory*-tallennustapa, joka tallentaa politiikan JavaScript-objektina muistiin.

Abac-modulilla politiikka luodaan *set_policy*-metodilla, jota kutsutaan sääntö kerrallaan. Ensimmäisenä parametrina säännölle annetaan tieto, mitä tallennustapaa se käyttää ja toisena parametrina säännön nimi. Kolmantena parametrina on funktio, jolla määritellään, milloin sääntö toteutuu, jolloin funktio palauttaa arvon *true* ja milloin ei toteudu, jolloin funktio palauttaa arvon *false* [Abac, 2014].

Pääsy tarkistetaan *can*-metodilla, jolle annetaan tallennustavan nimi ja säännön nimi. Näillä parametreilla Abac-moduli tunnistaa mitä sääntöä käytetään, koska kaikki säännöt

on määritelty sekä tallennustavan että säännön nimellä. Viimeisenä parametreina annetaan kaksi vastakutsufunktiota, *yes* ja *no*, joita kutsutaan, kun pyyntö sallitaan tai evätään, tässä järjestyksessä. Käyttäjän pääsyn sallimisen jälkeen *yes*-funktiossa annetaan pääsy objektiin ja *no*-funktiossa tehdään jotain sen jälkeen, kun pääsy evätään.

Accesscontrol-modulissa (www.npmjs.com/package/accesscontrol) politiikka määritellään ketjutetuilla metodeilla. Roolille annetaan oikeuksia *Grant*-metodilla, joka saa parametrina roolin nimen merkkijonona ja samalla se luo uuden roolin tai muokkaa jo olemassa olevaa. *Grant*-metodin jälkeen ketjutetaan oikeuksia funktiokutsuilla, joissa funktion nimi muodostuu jostakin CRUD-operaatiosta (create, read, delete tai update) ja siitä onko objekti subjektin omistama (*Own*) vai mikä tahansa (*Any*). Funktiolle annetaan parametrina objektin nimi ja mahdollisesti sallitut objektin kentät merkkijonoina [Accesscontrol, 2018].

Accesscontrol-modulissa pääsyoikeusolio saadaan *can*-metodilla, jolle myös annetaan parametriksi roolin nimi ja sille ketjutetaan oikeudet vastaavasti kuin *grant*-metodin kanssa. Pääsyoikeusolion *granted*-metodilla saadaan totuusarvo, jota voidaan käyttää ehtolauseissa. Olion *attributes*-metodilla saadaan ne objektin kentät, joihin roolilla on oikeus [Accesscontrol, 2018].

Kun halutaan tietää voiko subjekti poistaa omistamansa objektin, täytyy tarkistaa erikseen, onko subjekti oikeasti objektin tekijä, pelkkä *deleteOwn*-oikeuden käyttö ei sitä takaa. *Own* on pelkästään koodin lukemista helpottava osa, joka ei sinällään muuta logiikkaa. Myös objekteista käytetään itse olion sijasta niiden nimiä [Accesscontrol, 2018].

5.1.2 React

React (reactjs.org) on Facebookin kehittämä *front-end* -sovelluskehys, joka jakaa ulkoasua itsenäisiin komponentteihin, joita voidaan käyttää uudelleen. Komponentit ovat JavaScript-luokkia, jotka kirjoitetaan JSX-kielellä, joka laajentaa JavaScriptiä html-tyylisillä tageilla. JavaScriptia voidaan kirjoittaa JSX:n sisään aaltosulkeiden väliin. Komponentteja voidaan kutsua koodissa tageilla <*KomponentinNimi* /> ja komponentit voivat kutsua myös toisiaan [React, 2018].

Komponentit saavat parametreina ominaisuuksia (*props*), joilla voidaan antaa komponenteille muuttujia ja sitä kautta muodostaa dynaamista ja muuttuvaa käyttöliittymää. Kutsu <*KomponentinNimi message=”hei”* /> antaa komponentille ominaisuuden message, jonka arvo on merkkijono hei. Komponentit muodostavat perimyssuhteita, kun komponenteille annetaan lapsikomponentteja, jotka asetetaan sen tagien sisälle. Kutsuttu komponentti tulostaa lapset kohtaan, jossa on {props.children} [React, 2018].

Komponenteilla on myös yksityinen tila (*state*), joka sisältää muuttujia ja niiden arvoja. Tilan muuttujat eroavat ominaisuuksista siten, että niitä ei anneta muille komponenteille. Toisin kuin vakio-ominaisuudet, tilan objektit saavat muuttua, jolloin myös kom-

ponentin ulkoasun on tarkoitus muuttua, jolloin komponentti piirretään uudestaan. Tapahtumat (*events*) ovat funktioita, jotka muuttavat tilan arvoja, jolloin komponentin ulkoasu muuttuu. Esimerkiksi suljettavalla valikolla voi olla tilassa auki-muuttuja ja painikkeen painaminen laukaisee aukaise() -tapahtuman, joka muuttaa auki-muuttujaa ja komponenti aukaisee valikon. Myös tapahtumia voidaan antaa toisille komponenteille ominaisuuksilla [React, 2018].

React ylläpitää DOM-puusta virtuaaliversiota VirtualDOM ja se vertailee virtuaalisen ja oikean puun eroja. React pyrkii säästämään resursseja päivittämällä vain ne elementit, jotka oikeasti muuttuvat VirtualDOM:ssa eikä päivittämällä suoraan koko DOM-puuta. DOM-puuta ei käsitellä manuaalisesti [Krajka, 2015].

React-authorization-moduli tarjoaa roolipohjaisen pääsynhallinnan. Se sisältää komponentit *IfAllGranted ja IfAnyGranted*, joista ensimmäinen tarkistaa onko käyttäjällä kaikki tarvittavat roolit ja jälkimmäiselle riittää kaikista rooleista yksi sallittu rooli. Komponenteille annetaan sallitut roolit ja käyttäjän rooli(t) taulukoina. Pääsy sallitaan ja komponentille annettu lapsitagi näytetään, jos käyttäjällä on sallittu rooli tai roolit. Ominaisuudella *unauthorized* voidaan asettaa mikä tagi näytetään, jos pääsy evätään. *IfGranted* toimii yhdellä roolilla [React-authorization, 2017].

```
1  import React, { Component } from 'react';
2  import Unauthorizated from './Unauth.js';
3
4  import {IfAnyGranted} from 'react-authorization';
5
6  var user = {session_id: "QQ46Lp0bz0oxfyxU1our", role: 'ADMIN'};
7  var user2 = {session_id: "GQGCG4X6hzXJ6G7fXmy8", role: 'USER'}
8
9  class App extends Component {
10   render() {
11     return (
12       <IfAnyGranted
13         expected={['ADMIN','USER']}
14         actual={user.role}
15         unauthorizated={<Unauthorizated/>}
16       >
17         <div className="App">
18           <p>Pääsit tänne.</p>
19         </div>
20       </IfAnyGranted>
21     );
22   }
23 }
24
25 export default App;
```

Koodiesimerkki 2. React-authorization.

Koodiesimerkissä 2 rivillä 4 otetaan React-authorization-modulista käyttöön komponentti *IfAnyGranted,* jota käytetään rivillä 12 ja se suljetaan rivillä 20. Komponentille *IfAnyGranted* annetaan sallitut roolit ADMIN ja USER taulukkona ominaisuudessa *expected* ja käyttäjän rooli ominaisuudessa *actual.* Viimeisenä parametrina komponentille annetaan Unauthorized-komponentti, joka on otettu käyttöön tiedostosta Unauth.js rivillä 2 ja joka näytetään, jos käyttäjänpääsy evätään. Jos käyttäjän rooli on joko ADMIN tai USER, käyttöliittymässä näytetään komponentille riveillä 17-19 annetut lapsitagit.

5.1.3 Angular 5

Angular (www.angular.io) on Reactin tapaan komponentteihin perustuva *front-end* sovelluskehys. Google on kehittänyt Angularista lyhyessä ajassa useita versiota. Angular käyttää TypeScript-kieltä, joka laajentaa JavaScriptia tyypityksillä muuttujille ja rajapinnoilla olioille [Angular TypeScript Docs, 2018; TypeScript Interfaces 2019].

Komponentin metadata sisältää tiedot siitä, mikä tiedosto on komponentin oma html-malline ja minkä nimiseen tagiin komponentti asetetaan mallineessa. Data kulkee kaksisuuntaisesti mallineelta komponentille ja toisinpäin. Mallineessa sidotaan (*bind*) dataa komponenttien ja mallineiden välillä muuttujiin ja tapahtumiin (*action*). Esimerkiksi mallineessa painikkeen tapahtuma (*click*) sidotaan johonkin komponentin funktioon, joka käsittelee käyttäjän painikkeen painamisen. Komponentista annetaan mallineelle muuttujia ominaisuuksina (*property*), joita mallineessa voidaan näyttää aaltosulkeiden sisällä. Aaltosulkeiden sisällä voi olla putki {|}, jolla voidaan muotoilla näytettävää muuttujaa, jolloin putken vasemmalla puolella on varsinainen muuttuja ja oikealla puolella käytettävän putken tunnus ja sen muotoiluasetukset.

Itse komponentti on TypeScript-luokka, joka sisältää attribuutteja, rakentimen ja metodeita. Attribuutit ovat muuttujia ja olioita ja ne sisältävät tietoa, jota renderöidään mallineeseen. Metodit muuttavat attribuuttien dataa. Komponentit käyttävät rakentimina injektoituja palveluita (*service*), joita voidaan käyttää uudelleen. Palvelut vastaavat komponenttien luomisesta sekä datan käsittelystä kontrollerien tavoin. Itse komponenttien on tarkoitus olla MVC-mallin näkymiä. Angularin modulit ovat kokonaisuuksia, jotka sisältävät komponentteja, direktiivejä ja putkia jonkun asian toteuttamiseen [Angular Architecture Docs, 2018].

Angular Router tarjoaa reitityksen, jolla siirrytään näkymästä toiseen ja reitti näyttää tietyn komponentin. Reitit ovat käytössä sovelluksen modulissa *appRoutes*-listassa, jossa yhdelle reitille annetaan relatiivinen polku ja polkuun liitettävän komponentin nimi [Angular Router Docs, 2018].

Reiteille voidaan asettaa portteja (*guard*). Portit ovat metodeita, joista luodaan oma toteutus palvelussa ja jotka otetaan käyttöön sovelluksen reititysmodulissa tietyille reiteille. Portin logiikka toteutetaan itse palveluun metodin sisälle ja se palauttaa totuusarvon, jolla päätetään sisäänpääsystä reittiin.

Portilla *CanActivate* tuotetaan rakenne, jolla voidaan tarkastaa, onko käyttäjä kirjautunut sisään tai käyttäjällä on tietty rooli, jolla pääsee reittiin. *CanActivateChild* toimii samoin, mutta suojaa kaikkia reitin periviä lapsireittejä. Resurssia muokattaessa *CanDeactivate*-portilla toiminta voidaan pysäyttää tai peruuttaa objektin perusteella, joko varmistamaan käyttäjän toimia tai odottamaan kunnes reittiä voidaan kutsua asynkronisesti serverin ollessa valmis. [Angular Router Docs, 2018]

Tiedot käyttäjän roolista ja sisäänkirjautumisesta saadaan toteutettavan metodin sisään muilta palveluilta. Toteutettava portti voi esimerkiksi ottaa tiedon siitä onko käyttäjä kirjautunut sisään *authService*-palvelulta ja palauttaa sen perusteella *true* tai *false*. Toiset palvelut voivat palauttaa muita käyttäjän profiilin tietoja tai JWT-tokenin, joita käytetään portin toteutuksessa.

5.2 Python, Django

Python on alun perin skriptaukseen tarkoitettu ohjelmointikieli, joka lyhyytensä ja helpon ymmärrettävyytensä vuoksi on varsin suosittu aloituskieli ohjelmointiin. Ymmärrettävyyden tukemiseksi Pythonin rakenteissa käytetään mahdollisimman paljon luonnollisen englannin sanoja, kuten *and* ja *not* muissa ohjelmointikielissä käytettyjen merkkien ”&&” ja ”!” sijasta. Python pakottaa käyttämään koodin luettavuutta parantavia sisennyksiä, joilla koodilohkot erotellaan rakenteellisesti muiden ohjelmointikielten aaltosulkeiden tapaan. Pythoniin voidaan ladata kirjastoja (*package*), valmiiksi ohjelmoituja kokonaisuuksia, jotka lisäävät erilaisia toimintoja. Yleisin Python-tulkki on c-kielellä kirjoitettu Cpython, joka kääntää Python-lähdekoodin tavukoodiksi ja suorittaa sen.

Django (www.djangoproject.com) on web-sovelluskehys, joka on kirjoitettu Python-kielellä. Djangossa näkymä (*view*) on Python-funktio, joka ottaa parametrina HTTP-pyyntöä vastaavan objektin *request* ja palauttaa HTTP-vastauksen. Nimestään huolimatta näkymät vastaavat pikemminkin MVC-mallin sovelluskehysten kontrollereita (*controller*), koska ne sisältävät datan hakemisen ja lähettämisen mallineille sekä mahdollisen esikäsittelyn. Näkymät on listattu Djangossa oletuksena ”views.py”-tiedostoon. Näkymät antavat datan mallineille muuttujina *context*-sanakirjassa. Mallineissa muuttujat injektoidaan sivulle tuplahakasulkeiden sisällä ja {% %} -tageilla voidaan esittää loogisia rakenteita, kuten if-lauseita tai periyttää toinen malline [Django Software Foundation, 2018].

Pythonissa moduli on tiedosto, joka sisältää muuttujien ja funktioiden määrittelyjä ja lausekkeita. Djangon sisäänrakennettuja moduleita, jotka toteuttavat eri web-sovelluskehyksen toimintoja, löytyy erityisesti ’django.contrib’-nimiavaruudesta. Muita ’django-contrib’-nimiavaruudesta valmiina löytyviä kirjastoja ovat mm. hallintasivu ylläpidolle, luokka uudelleenohjauksille, sessionhallinta sekä rajapinta *Postgresql* -tietokannalle.

Djangon ominaisuuksia otetaan käyttöön lisäämällä ja poistamalla niitä ’settings.py’ -konfigurointitiedostossa. Konfiguraatiotiedosto on oikeastaan vain tavallinen Python-moduli, jossa asetukset ovat listoja ja listojen nimet kirjoitetaan isolla alkukirjaimella.

5.2.1 Auktorisointi Djangossa

Djangon autentikaatio- ja pääsynhallintamoduli *auth* otetaan käyttöön lisäämällä 'django.contrib.auth' konfiguraatiotiedoston INSTALLED_APPS-listaan [Django Software Foundation, 2018].

Jotta autentikaatio toimisi, tulisi myös kirjaston väliohjelmisto 'auth.middleware.AuthenticationMiddleware' olla lisättynä konfiguraation MIDDLEWARE-listaan. Listassa käyttöön otetut väliohjelmistot käsittelevät pyyntöobjektia (*request*) vuorotellen, ennen kuin pyyntö päätyy näkymälle ja autentikaatio-väliohjelmisto lisää pyyntöobjekteihin käyttäjäolion *user*. Jos käyttäjä on kirjautunut sisään, käyttäjäolio sisältää tietokantaan tallennetut käyttäjän tiedot, kuten käyttäjänimen ja hajautuksella salatun salasanan. Jos käyttäjä ei ole kirjautunut sisään, käyttäjäolio on tyhjä [Django Software Foundation, 2018].

Django sisältää malliobjekteja (*Model*), joiden kautta käsitellään tietokantatauluja ORM-menetelmällä [Django Software Foundation, 2018]. ORM-menetelmällä (Object-Relational mapping) muutetaan ohjelmakoodissa käsiteltävät tavalliset objektit tietokannan objekteiksi ja toisinpäin [Butterfield and Ngondi, 2016]. Kun koodissa luodaan malliobjekti, jolle on annettu lista attribuuteista, Django luo malliobjektia vastaavan tietokantataulun, jossa attribuutit ovat taulun sarakkeita. Kun malliobjektista luodaan olio, Django lisää tietokantaan automaattisesti uuden rivin. Tietokantaa siis käytetään ilman SQL-kyselykieltä objektien avulla.

Myös käyttäjä on malliobjekti, jonka tiedot on tallennettu tietokantaan. Oikeus- (*Permission*) ja käyttäjä -malliobjektit löytyvät *auth.models*-nimiavaruudesta. Djangon pääsynhallinta lisää automaattisesti jokaiselle malliobjektille, mitä ohjelmassa käytetään, lisää, poista ja muokkaa -oikeudet. Sisäänkirjautumattoman käyttäjän oliolle, jonka nimi on AnonymousUser, voidaan asettaa omat oikeutensa [Django Software Foundation, 2018].

Käyttäjän oikeuksia voidaan tarkastaa näkymissä pyynnöstä saadun käyttäjäolion *has_perm* -funktiolla, jolle annetaan oikeuden nimi ja jonka palauttaman totuusarvon mukaisesti toiminto voidaan evätä tai sallia. Käyttäjällä on *user_permissions*-attribuutti, joka sisältää käyttäjän oikeudet ja oikeuksia voidaan asettaa koodissa sen *add*-metodilla ja poistaa *remove*-metodilla. Käyttäjä voi kuulua yhteen tai useampaan ryhmään (*Group*-malliobjekti), jolle asetetut oikeudet periytyvät ryhmän yksittäisille käyttäjille, jonka vuoksi käyttäjän oikeuksia ei tarvitse tarkistaa ryhmän tasolla [Django Software Foundation, 2018].

Mallineiden puolella kirjautuneen käyttäjän oikeudet voidaan kaivaa esiin *perms*-muuttujasta ja sen sovelluksen nimen attribuutista [Django Software Foundation, 2018]. Kontrollitageilla voidaan sen avulla ohjata mitä sivulla näkyy. Esimerkiksi *if*-tagissa voidaan tarkistaa, onko oikeus johonkin ominaisuuteen käyttäjän oikeuksissa, jolloin tagin

sisällä sisällytetään kyseinen ominaisuus. Niille, joilla oikeutta ei ole, ominaisuus ei näy tai heille näytetään jotain muuta *else*-tagin sisällä. Myös sen, onko käyttäjä kirjautunut sisään, voi tarkistaa mallineessa käyttäjäolion *is_authenticated*-metodista, jolloin käyttäjälle voidaan näyttää tilanteesta riippuen joko ”kirjaudu sisään”- tai ”kirjaudu ulos”-linkki.

5.3 Java, Spring

Java on ohjelmointikieli staattisella tyypityksellä, jonka kehitti Sun Microsystems 1990-luvun alussa. Kielen omistaa nykyään Oracle Corporation. Java on rakennettu olio-ohjelmointia silmällä pitäen [Butterfield and Ngondi, 2016].

Javan syntaksi perustuu C++-kieleen, mutta se sisältää enemmän sisäänrakennettuja luokkia ja ominaisuuksia, jotka on tehty oliopohjaisesti ja vastaavasti matalan tason ominaisuuksia on vähemmän. Esimerkiksi C++:n poikkeukset ovat Javassa toteutettu olioilla ja C++:n suora muistinhallinta on abstraktoitu pois. Java-tiedostot käännetään tavukoodiksi kääntäjällä ja ajetaan Jvm-virtuaalikoneessa. Ohjelmointiympäristö Java System Development Kit sisältää kirjastot, *javac*-kääntäjän, ajo-ohjelman *java* sekä *jar*-pakkauksenhallinnan [Van Hoff, 1997; Oracle, 2014].

Alun perin Java on tarkoitettu pieniin web-sovelmiin (*applet*), jotka viestivät verkon välityksellä [Butterfield and Ngondi, 2016]. Perinteisesti Javalla on ohjelmoitu isoja tietojärjestelmiä ja web-sivujen serveripuolta. Nykyään sitä käytetään Android-sovellusten koodaamiseen.

5.3.1 Spring MVC

Spring MVC eli Spring (www.spring.io) on suosittu ja pitkään (vuodesta 2002) käytössä ollut web-sovelluskehys. Springissä *dispatcher-servlet* ohjaa pyynnöt kontrollereille. *Dispatcher* vastaa siis URL-osoitteiden reitityksestä eli tietty pyynnöstä saatu relatiivinen osoite on määrätty ohjaamaan tietylle kontrollerille. Reititykset asetetaan kontrolleriluokissa kontrollerimetodin yläpuolelle @RequestMapping -annotaation avulla. Esimerkiksi kun kontrollerin *HomeController* yläpuolella on @RequestMapping(“/home”), *dispatcher* ohjaa polun home sisältävät pyynnöt tähän kontrolleriin [Amuthan and Aniket, 2014].

Sovelluksen konteksti (*Application Context*) on kontti (*container*), joka luo ja ylläpitää sovelluksen olioita (*bean*), sekä pitää huolen olioiden välisistä riippuvuussuhteista. Kontrolleri luo näkymä- ja kontrolleriluokista oliot ajon aikana. Joidenkin olioiden tehtävä on ohjata pyyntöjä *dispatcher*:ltä kontrollereille ja näkymäselvittäjäolio (*ViewResolver)* osaa kertoa *dispatcher*:lle mikä näkymä käyttäjälle näytetään. Konteksti konfiguroidaan XML-merkkauskielellä tai annotaatioilla [Amuthan and Aniket, 2014].

Springin avulla tehdyllä web-sovelluksella on hierarkkisia tasoja (*layer*) tiedon tallennukseen, muokkaamiseen ja esittämiseen. *Domain*-taso on lähimpänä tietokantaa ja

kaukaisimpana käyttäjästä, se sisältää tietomallit tietokannan tauluista eli oliot, joiden kautta käsitellään tietokannan tauluja. *Persistence*-tasoon ohjelmoidaan tietomallien käyttöön tarvittava logiikka, kuten CRUD-operaatiot, joilla voidaan esimerkiksi luoda (*Create*) uusi malli. *Service*-tasolle tehdään monimutkaisemmat sovelluksessa tarvittavat operaatiot, jotka voivat käyttää useampia *persistence*-tason metodeita. Lähimpänä käyttäjää sijaitseva *Presentation*-taso esittää tiedot ja käyttöliittymän, se sisältää edellä mainitut kontrollerit, näkymät, *dispatcher*:n ja näkymänselvittäjän [Amuthan and Aniket, 2014].

Auktorisointia hoitaa *Spring Security* -kirjasto. Osoitetasolla pääsynhallinta toimii kontekstin konfiguraatiossa, johon pannaan *intercept-url* -asetuksia, jolla tietyt URL-osoitteet vaativat sisäänkirjautumisen. Xml-muodossa *<intercept-url pattern=”/polku/” access=IS_AUTHENTICATED_FULLY>* määrittää “/polku”-osoitteen vaativan autentikaation, jolloin sovellus päästää vain sisäänkirjautuneet käyttäjät kyseiseen osoitteeseen ja uudelleenohjaa sisäänkirjautumattomat sisäänkirjautumissivulle. Attribuutilla *requires-channel* voidaan rajata protokollaksi HTTP tai HTTPS.

Access-parametrilla hallitaan kuka voi päästä url-osoitteeseen, ei-auktorisoiduille reiteille annetaan parametriksi arvo IS_AUTHENTICATED_ANOMYOUSLY jolloin sisäänkirjautumista ei tarvita. Access-parametrilla IS_AUTHENTICATED_REMEMBERED voidaan käyttää pitkäaikaisia “pysy sisäänkirjautuneena”-evästeitä, joilla käyttäjän ei tarvitse kirjautua joka kerta. Pakollinen autentikaatio määritellään *access*-attribuutin IS_AUTHENTICATED_FULLY-parametrilla, kuten edellä [Jagielski and Nabrdalik, 2013].

Url-osoitteisiin voidaan asettaa myös roolipohjaista pääsynhallintaa. Luvan saava käyttäjärooli määritellään access-parametrin ROLE_ROOLINNIMI-arvolla, jossa ROLE on etuliite rooleille. Käyttäjärooli on käytännössä merkkijono UserDetails-objektissa ja tietokannassa. Access-parametrin arvot voivat myös olla Springin merkkauskielen SPEL-funktioita, kuten *hasRole(ROLE_ROOLINNIMI)*, *hasAnyRole(ROLE1,ROLE2)*, *isAuthenticated()* tai *isAnonymous()* sekä attribuutteja *denyAll* ja *permitAll*, joilla estetään tai sallitaan kaikki käyttäjät [Jagielski and Nabrdalik, 2013].

Metoditasolla pääsynhallintaa käytetään annotaatioilla eli koristajilla, joille annetaan parametrina sallitut tai evätyt subjektit SPEL-merkkauskielellä. Yleisimmin käytetty annotaatio on *@PreAuthorize()*, esimerkiksi *@PreAuthorize(hasRole(ROOLI))* sallii metodikutsun roolille nimeltä ROOLI, se siis rajaa pääsyä ennen metodia. Vähemmän käytetty *@PostAuthorize(hasRole(ROOLI))* estää tai sallii pääsyä metodin kutsumisen jälkeen. Annotaatiot lähettävät pääsynhallinnan epäonnistuessa AccessDeniedException -objektin [Jagielski and Nabrdalik, 2013].

Nykyinen sisäänkirjautunut käyttäjä löytyy *authentication* -attribuutista. SPEL-merkkauskielellä voidaan esittää monimutkaisempia rakenteita, kuten tarkistaa annotaation

sulkeiden sisällä onko käyttäjänimi *authentication.name* sama kuin objektin omistaja, jonka annotaatio ottaa auktorisoidulle metodille annetusta parametrista [Jagielski and Nabrdalik, 2013].

@PostAuthorize/Filter ja *@PostFilter* -annotaatiolla, eli kun annotaatiossa on lisäksi */Filter* -lisämääre, voidaan metodin palauttamaan olioon tai listaan päästä käsiksi ja suodattaa oliota tai listaa SPEL-rakenteen avulla. Yleensä suodatusta käytetään rajaukseen eli listasta voidaan palauttaa vain tietyt alkiot ja oliosta tietyt parametrit. Annotaatiolla voidaan siis rajata metodin palauttamaa listaa: esimerkiksi sillä voidaan poistaa muut kuin subjektin omistamat alkiot [Jagielski and Nabrdalik, 2013].

5.4 C#, ASP.Net

C# (“*c sharp”)* on olio-ohjelmointikieli, jonka syntaksi perustuu C++ -kieleen ja jossa on samoja ominaisuuksia kuin Javassa [Butterfield and Ngondi, 2016]. Toisaalta koska C++ ja Java ovat keskenään syntaksiltaan samanlaisia, C#:n voidaan sanoa perustuvan Javaan. Molemmat ovat vahvasti oliopohjaisia. C# käännetään Javan tavoin välikielelle tavukoodiksi, jota ajetaan CLR-virtuaalikoneessa. Molemmissa on Object-luokka, josta kaikki luokat periytyvät. Roskankeruu on kummassakin automaattista ja C++:n moniperinnästä on luovuttu [Bagnall et al., 2002].

C# on osa Microsoftin .NET -alustaa (www.microsoft.com/net/), joka sisältää kehitysympäristön ja FCL-kirjaston (*Framework Class Library*). Kirjastoa voidaan käyttää myös Visual Basic ja C++ -kielillä, koska myös ne käännetään .NET:ssä CIL-tavukoodiksi. FCL-kirjasto toteuttaa samoja toimintoja kuin C++- ja Java-kielten kirjastot sisältävät [Bagnall et al., 2002].

Microsoftin Visual Studio -ohjelmointiympäristö on tarkoitettu .NET-alustan ohjelmointiin. Asp.Net on .NET -alustan osa web-sovellusten kehitykseen. ASP.Net Core on uusin versio ASP.Net:stä, joka on julkaistu 2016 ja pitkälti uudelleenkirjoitettu.

ASP.Net käyttää MVC-mallia. Kontrollereissa data asetetaan ViewData-sanakirjaan, jotka saadaan näkymissä käyttöön View()-metodilla. Näkymät on jaettu automaattisesti polkuihin /Kontrollerin_nimi/Views/Näkymän_nimi/, joihin URL asetetaan viittaamaan. Näkymissä käytetään *Razor*-merkkauskieltä, joissa funktiot ja muuttujat tunnistetaan @-merkeillä [Walther et al., 2008].

5.4.1 Pääsynhallinta ASP.Net:ssa

ASP.Net Core:ssa sisäänpääsyt tarkistetaan Authorize-koristajalla (*attribute*), joita asetetaan kontrollereissa metodin eteen tai koko kontrolleriluokan eteen, jolloin se vaikuttaa kaikkiin sen metodeihin. Jos Authorize asetetaan molempiin paikkoihin, yksittäisen metodin eteen asetettu attribuutti tarkentaa koko luokan eteen asetettua. Kontrollerin metodit ovat toimintometodeita (*action-method*), jotka manipuloivat dataa ja palauttavat lopuksi

näkymän. C#:n koristajat vastaavat Javan annotaatioita. Ne merkitään hakasulkujen sisään ja niille voi asettaa parametreja. Jos Authorize-koristajalla ei ole parametreja, metodiin tai metodeihin pääsyyn vaaditaan sisäänkirjautuminen. *Roles*- ja *Policy* -parametreilla rajataan pääsy metodiin tietyillä rooleille tai säännöille [Anderson et al., 2018].

AuthorizationService -luokalla pääsynhallinta saadaan käyttöön muuallakin kuin kontrollereissa. Kun näkymissä käytetään @inject-direktiiviä AuthorizationService -luokalle, jolla luodaan riippuvuusinjektio auktorisoinnille eli periytetään luokan ominaisuudet, pääsynhallintaa voidaan käyttää käyttöliittymän muokkaamiseen näkymässä. Tällöin ehtodirektiivejä @if ja @else voidaan asettaa ennen piilotettavia tai näytettäviä käyttöliittymän osia. Ehtodirektiiveille annetaan AuthorizeAsync-metodilla käyttäjäolio sekä säännön nimi ja se palauttaa totuusarvon, jolla direktiivin sisältämät elementit piilotetaan tai näytetään [Anderson et al., 2018].

Yksi palvelu on luokka, joka toteuttaa jonkun toiminnon, mikä toimii usein internetin kautta eli sen metodeja käytetään HTTP pyynnöillä ja vastauksilla. Palvelut kommunikoivat keskenään XML-tiedostoilla SOAP-protokollalla. Web-sovelluksen palveluihin voi kuulua esimerkiksi palvelut sisäänkirjautumiseen, tietokantayhteyteen ja käyttäjänhallintaan. Sovelluksen kaikki palvelut asetetaan konfiguraatiossa palvelukokoelmaan *services*, joka on IServiceCollection-rajapinnan instanssi. Palvelukokoelmaan asetetaan uusia ominaisuuksia riippuvuusinjektiolla (*dependency injection*), jolloin tiukan objektireferenssin sijasta tietyntyyppiseen palveluun kytketään kyseisen ominaisuuden toteuttava rajapinta. [Bagnall et al., 2002; Smith and Scott, 2016].

Politiikan säännöt asetetaan palvelukokoelman AddAuthorization-metodilla ConfigureServices-funktiossa, jolla määritellään palveluiden asetukset ja joka sijaitsee Startup.cs tiedostossa. AddAuthorizationille annetaan anonyymi funktio, joka saa parametrina options-olion. Politiikan säännöt määritellään funktion sisällä *options*-parametrin AddPolicy -metodikutsuilla, joihin annetaan säännön nimi ja vaadittu kriteeri. Vaadituksi kriteeriksi asetetaan joku *policy*-luokan metodi, kuten *RequireRole*, jolle annetaan oikeuteen tarvittava rooli tai *addPolicy*, jolla luodaan uusi sääntö, joka toteuttaa *IAuthorizationRequirement* -rajapinnan [Anderson et al., 2018].

Kriteeriksi voidaan asettaa myös vaatimus (*claim*), joka on nimi-arvo-pari, mikä edustaa käyttäjän ominaisuuksia, kuten henkilön ”Matti Meikäläinen” syntymävuosi. Asp.Netin vaatimukset vastaavat attribuuttipohjaista pääsynhallintaa. Vaatimukselle joko annetaan politiikan määrittelyssä sallitut arvot tai tarkastetaan, onko vaatimus ylipäätään olemassa käyttäjäluokassa.

ASP.Net:stä löytyy myös resurssipohjainen pääsynhallinta, jossa subjekti päästetään objektiin resurssin perusteella ja objekti yleensä sisältää tiedon millä perusteella sitä voi käyttää. Tarkistus voidaan myös tehdä toisin päin eli onko käyttäjällä oikeus objektiin. Molemmissa tapauksissa objekti tulee hakea ennen kuin oikeudet tarkistetaan, joten

Authorize-koristajaa ei voi käyttää, koska attribuutit evaluoidaan ennen kuin tiedot haetaan funktiossa. Sen sijaan auktorisointi tapahtuu kontrollerin sisällä, johon periytetään IAuthorizationService -rajapinta, joka sisältää myös toisentyyppisen AuthorizeAsync-metodin, jolle annetaan käyttäjän lisäksi resurssiobjekti ja kriteerit. Metodi tarkistaa täyttääkö käyttäjä kaikki kriteerit ja päästääkö se käyttäjän käsiksi resurssiin. Kriteerit annetaan listana luokkia, jotka toteuttavat IAuthorizationRequirement-rajapinnan. Yksi kriteeri voi olla esimerkiksi sellainen, että käyttäjän tulee omistaa resurssi ja metodissa tarkistetaan, onko resurssiin merkitty tekijän tunnus sama kuin sitä käyttävä subjektin tunnus [Anderson et al., 2018].

5.5 Ruby, Ruby on Rails

Ruby on 1995 julkaistu ohjelmointikieli, joka sisältää vaikutteita kielistä Perl, Eiffel, Smalltalk ja Lisp. Ruby on syntaksiltaan lyhyttä ja yksinkertaista muistuttaen pseudokoodia, mutta on silti ilmaisuvoimaista, jolloin sillä voidaan tehdä myös monimutkaisia ohjelmia. Ruby on dynaamisesti tyypitetty ja siihen on yhdistetty oliopohjaisten, imperatiivisten ja funktionaalisten ohjelmointikielten syntaksia. Rubyn suosio on kasvanut erityisesti Ruby on Rails -kehyksen vuoksi [Butterfield and Ngondi, 2016; Lewis, 2014].

Ruby on Rails (www.rubyonrails.org/) käyttää konventio-konfiguraation-sijasta -filosofiaa (*convention over configuration*), joka tarkoittaa, että Railsissa on määritelty websovelluksen rakennetta ja nimeämiskäytäntöjä hyvin pitkälle jo valmiiksi ja omien asetusten käyttämistä ei suosita. Railsin käyttämä tiedostorakenne, web-serveri, kirjastot ja tietokantataulujen nimet annetaan valmiiksi ja niiden muuttaminen on tehty hankalaksi. Toinen filosofia on toiston välttäminen: Railsissa pyritään siihen, että jokaisesta toiminnosta on vain yksi määritys ja se sijaitsee vain yhdessä paikassa. Metodeita ja vakioita sitten linkitetään (*map*) moniin eri tiedostoihin ja periytetään saman komponenttityypin sisällä [Lewis, 2014].

Ruby on Rails noudattaa orjallisesti MVC-mallia, jonka tiedostot sijaitsevat aina *app*-kansiossa. Näkymien *Embedded Ruby*-kielessä html-koodissa voidaan käyttää Ruby-koodia <%>-tagien sisällä, joka toiston eliminoimiseksi voidaan siirtää kontrollereille *helper*-kansioon apurimetodiksi (*helpers*). Apurit ovat kontrollerin omistamia metodeita, joita voidaan siis käyttää näkymissä. Ruby on Rails:ssa on määritelty valmiiksi jonkin verran apureita, mutta niitä voidaan luoda omista kirjastoista.

Railsissä on pääkontrolleri *application_controller.rb*, josta muut kontrollerit perivät asetuksia. Railsin mallit toteuttavat automaattisesti *ActiveRecord* tietomallia ja ORM:a, jonka vuoksi niillä on oletuksena CRUD-operaatiot.

Rubyn kirjastot ovat metodien ja vakioiden kokoelmia, jotka muistuttavat luokkia, mutta toimivat kuten nimiavaruudet. Sekä Rubyn kirjastoja, että pakettienhallintaohjelmaa kutsutaan nimellä gem.

5.5.1 Pundit

Eräs Ruby on Rails:n kirjasto (*gem*) pääsyhallintaan on yksinkertainen Pundit (www.github.com/varvet/pundit), joka otetaan käyttöön lisäämällä se pääkontrolleriin, josta se periytyy myös muihin kontrollereihin. Pundit keskittyy politiikkaluokkiin, jotka ovat tavallisia Ruby:n luokkia, joille annetaan parametrina käyttäjä sekä malliobjekti, mikä tekee sen pääsynhallinnasta resurssikohtaista. Politiikkaluokat sijaitsevat app/policies -kansiossa.

Politiikkaluokka toteuttaa myös jonkun metodin "lupa?", joka on varsinainen sääntö ja se linkittyy samannimiseen kontrollerin kyselymetodiin. Mallille nimeltä Luokka1 määritellään siis politiikkaluokka Luokka1Policy, jonka jälkeen kontrollerin kyselymetodin sisälle voi laittaa *authorize*-avainsanan, jonka kohdalla samanniminen kyselymetodi tarkistaa pääsyn politiikkaluokan samannimisellä metodilla [Pundit 2018].

Näkymissä mallin Luokka1 pääsyoikeus voidaan tarkistaa politiikkaluokan lupa-metodilla <%= if policy(@luokka1).lupa? %>. Politiikkaluokan lupa-metodin kysymysmerkki tarkoittaa metodin palauttavan totuusarvon, jonka if-käsittelee [Pundit, 2018].

5.6 PHP, Laravel

PHP (www.php.net/) on vuonna 1995 julkaistu skriptauskieli, joka on tarkoitettu web-kehitykseen [Butterfield and Ngondi, 2016]. Sen tyypitys on dynaamista ja heikkoa, eikä PHP:stä ollut alkuaikoina virallista standardia. PHP-kielellä ohjelmoiduissa web-kehyksissä ORM ja tietokannan kanssa viestiminen on yleensä pitkälle kehitettyä ja abstraktoitua, jolloin samat oliot toimivat automaattisesti monilla eri tietokannoilla.

Web-sovelluskehys Laravel (www.laravel.com) sisältää tuen ladattaville kirjastoille, mutta siinä on myös paljon sisäänrakennettuja ominaisuuksia. Laravel toteuttaakin konventio-konfiguraation-sijasta -filosofiaa (*convention over configuration*), jolloin se päättää asioista ohjelmoijan puolesta. Kirjastot ja Laravelilla toteutetut web-sovellukset toimitetaan paketteina (*bundle).* Laravelissa on muiden PHP-kehysten tapaan ajureita (*driver*), esimerkiksi session hallintaan, joiden toimintaa voidaan muuttaa ilman, että niiden käyttäminen muuttuu. Tietokantamigraation oliot toimivat samalla tavalla riippumatta tietokanta-alustasta. [Otwell, 2018; McCool, 2012]

Laravelin komentorivityökalulla *Artisan*:lla voi luoda ja asentaa paketteja, antaa palvelimelle ajastettuja tehtäviä sekä luoda malleja. Laravel käyttää oletuksena Blade-mallineita [McCool, 2012].

5.6.1 Pääsynhallinta Laravelissa

Auktorisointi on Laravelissa sisäänrakennettuna, sen toiminta perustuu portteihin (*gate*) ja politiikkaan (*politic*), jotka molemman rekisteröidään *AuthServiceProvider*-luokassa. Portit ovat sulkeumia reititysfunktioille, jotka tarkistavat pääseekö käyttäjä funktioon. Portit määritellään Gate::define -metodilla, jolle annetaan parametreina reitin nimi sekä

takaisinkutsufunktio, joka määrittää totuusarvon palauttavan tarkistuksen. Takaisinkutsufunktio saa parametreina esimerkiksi käyttäjän ja tarkistettavan objektin [Otwell, 2018].

Takaisinkutsufunktio voi olla myös jossain muualla toisen luokan sisällä, jolloin se annetaan muodossa Luokka@metodinNimi. Gate::recource(Luokka) -kutsulla voidaan asettaa auktorisointi automaattisesti kaikkiin toisessa tiedostossa olevan luokan metodeihin.

Reititysfunktioiden sisällä portteja käytetään *allows* ja *denies* -metodeilla, joista *allows* palauttaa arvon tosi, kun pääsy sallitaan ja *denies* arvon tosi, kun pääsy evätään ja arvoilla voidaan päättää mitä tehdään sen jälkeen. Käyttäjäkohtaisesti voidaan käyttää forUser-metodia, jolle annetaan käyttäjä, jonka jälkeen ketjutettu *allows* tai *denies* -metodi tarkastelee kyseisen käyttäjän pääsyoikeutta. Tärkeä sääntö voidaan määritellä *before*-metodiin, joka tarkistaa oikeuden ennen muita portteja [Otwell, 2018].

Laravelissa politiikat ovat luokkia, jotka tarjoavat pääsynhallinnan mallikohtaisesti. Politiikat tulee rekisteröidä *AuthServiceProvider*-luokan policies -attribuuttiin arvolla *Malli::class => MallinPolitiikka::class*, jossa malli linkitetään sen politiikkaan.

Itse politiikkaluokkaan asetetaan tarkastusmetodeita, jotka tarkistavat sisäänpääsyn ja palauttavat totuusarvon samalla tavoin kuin portteihin asetettavat takaisinkutsufunktiot. Tarkastusmetodille ei ole pakko antaa objektia vaan pelkkä käyttäjä, jos pääsynhallinta ei ole resurssikohtaista. Laravelin käyttäjäluokassa on valmiiksi *can* ja *cant* -metodit, jotka palauttavat totuusarvon siitä, voiko käyttäjä käyttää mallia. Myös politiikkojen kanssa voidaan käyttää *before*-metodia, joka tarkastaa oikeuden priorisoidusti ennen varsinaista politiikkaa [Otwell, 2018].

Tarkistusmetodien sijaan politiikkojen kanssa voidaan käyttää myös väliohjelmistofunktioita, jotka otetaan käyttöön reititysten yhteydessä. Ne tarkastavat sisäänpääsyn ennen kuin pyyntö menee reititysfunktioihin. Väliohjelmistolle annetaan can-attribuutti ja auktorisoitavan metodin nimi sekä mahdollisesti mallin luokka. Auktorisointi voidaan tehdä myös kontrollerissa *authorize*-apurilla, jolle annetaan auktorisoitava metodi ja malliolio [Otwell, 2018].

Politiikat rekisteröidään mallikohtaisesti luokilla, joten niitä on luonnollista käyttää silloin kun pääsynhallinta on resurssipohjaista. Portit toimivat reitityksessä, joihin ei liity malleja. Laravelin Blade-mallineissa voidaan käyttää @can, @else, @unless ja @cannot -direktiivejä haluttujen elementtien näyttämiseen ja piilottamiseen. Ne vastaavat erilaisia if ja else -lauseiden yhdistelmiä.

5.7 Yhteenveto

Tässä kohdassa vertaillaan edellisissä kohdissa esiteltyjen sovelluskehysten auktorisointeja keskenään.

Laravelin pääsynhallinta on kaikista monipuolisinta, koska se on toteutettu monella eri suunnittelumallilla, mutta se aiheuttaa myös valinnan vaikeutta. Laravelistä löytyvät

apurit, koristajat sekä tarkistusmetodeita, jotka voivat sijaita käyttäjäoliossa, politiikan luokassa tai portissa. Varsinaisesti yksipuolista ratkaisua ei löydy mistään kehyksestä.

Laravelin politiikat ovat samanlaisia kuin .Net:n resurssipohjaiset politiikat, koska molemmissa politiikat tulee rekisteröidä tietylle luokalle autentikaatiopalvelussa ennen kuin ne voidaan ottaa käyttöön ja itse politiikalle tulee kirjoittaa oma luokka tai kaksi. Laravelin politiikkoja on kuitenkin helpompi käyttää ja politiikkaluokka on yksinkertaisempi kuin ASP.Net:ssä, se sisältää vain vertailuja tekeviä metodeita, jotka saavat käyttäjän ja mallin parametreina. Molemmissa voi tehdä monimutkaisia tarkistuksia käyttäen mallin ja käyttäjän luokkien jäseniä.

Sekä Angularissa että Laravelissa on portteja, jotka muistuttavat toisiaan. Molemmissa portit määritellään reittikohtaisesti ja niille tehdään tarkistusmetodi, joka palauttaa totuusarvon, jonka perusteella reittiin pääsee tai ei pääse sisään. Kummassakin kehyksessä on autentikaatiopalvelu, jonne portit määritellään.

Koristajia on vain kehyksissä Asp.Net, Spring ja Django. Sekä Asp.Netissa että Springissa koristajia käytetään samalla tavalla käyttäjän roolin tai attribuutin tarkistuksiin. Javan koristajassa *isFullyAuthenticated* -metodilla tarkastetaan, onko käyttäjä sisäänkirjautunut, mutta .Netissa siihen riittää pelkkä tyhjä koristaja, joka on lyhyt kirjoittaa, mutta sen tarkoitus ei helposti selviä koodia lukevalle. Muuten Asp.Netin koristajat ovat ulkonäöltään selkeitä ja niitä voi käyttää attribuuttien, kriteerien ja roolien kanssa. Djangossa löytyy vastaava koristaja *@login_required()*. Springin koristajat ovat monipuolisimpia, koska niillä voidaan itse kirjoittaa monimutkaisia vertailuja SPEL-kielen avulla, jolloin niitä voidaan käyttää tarkistusmetodien sijasta, joita muissa kehyksissä tarvitaan monimutkaisempaan pääsyn määrittelyyn. Djangon koristajat muistuttavat valmiiksi määriteltyjä funktioita, kun taas Asp.Net:ssä ja Javassa koristajien logiikka tehdään enemmän itse koristajassa. Djangon koristajille annetaankin funktioiden tapaan parametreja, joko itse tehty tarkistusfunktio tai luvan nimi.

Politiikka määritellään useimmissa kehyksissä lisäämällä yksittäinen politiikka olion metodilla, missä olio on joko pääsynhallintaluokan tai käyttäjän instanssi. Djangossa politiikka on vain lista oikeuksista merkkijonoina, joka sijaitsee käyttäjäoliossa ja jonne voidaan lisätä uusi oikeus *add*-metodilla. Node-moduleissa Abac ja Node-abac kutsutaan pääsynhallintaluokan tarjoamaa oliota, jos halutaan lisätä uusi sääntö. Joissain kehyksissä politiikkaluokka tehdään itse, ja jos näin on, sen sisälle tehdään yleensä pääsyoikeuksien tarkistusmetodit. Joissain Node-moduleissa, kuten Node-abac:ssa, politiikka määritellään JSON-muodossa, joka sisältää subjektit tai objektit ja niiden oikeudet merkkijonoina. Pidän politiikan luomista JSON-muodossa vaikeaselkoisena. Springissä politiikka voidaan asettaa XML-muodossa koskemaan tiettyä url-osoitetta.

Vaikka Django tarjoaa valmiita metodeita oikeuksien tarkistukseen, joutuu niiden tarkistuslogiikkaa kirjoittamaan sinne missä metodeita käytetään, joko monimutkaisiin if-

lauseisiin tai antamaan logiikan metodille parametrina, mikä tekee koodista sekavaa. Laravelissa tarkistuslogiikan tekevät metodit kirjoitetaan pääsynhallintaluokan sisälle ja koodissa kutsutaan luokan yleistä *can*-metodia oikeuden nimellä, mikä on siistimpää. Pidän tarkistusmetodien tekemistä kootusti luokkaan parempana ratkaisuna, jolloin niitä voidaan käyttää uudelleen useammassa paikassa. Node.js-moduli Abac:ssa tarkistusfunktio annetaan säännön lisäämisen yhteydessä pääsynhallintaoliolle takaisinkutsufunktiona, mikä sekin on selkeä ratkaisu.

Djangon sisäänrakennettu pääsynhallinta on laajempi kuin yksi Node.js-moduli keskimäärin, koska se sisältää sekä attribuutit että ryhmät sekä useampia eri tapoja tarkistaa pääsy. Koska Noden auktorisointimodulit asentuvat paketinhallinnan avulla, niiden käyttö on helpompaa kuin Djangossa, jossa joutuu käsin muuttamaan konfiguraatioita.

Front-end -puolella sekä Angularissa että Reactissa käyttöliittymää pilkotaan hierarkkisiin komponentteihin, jolloin molempien kehysten pääsynhallinta keskittyy komponentin näyttämisen estämiseen jollain tavalla. Kehykset eroavat toisistaan siten, että Angularissa auktorisointiin on yksi sisäänrakennettu ratkaisu, mutta Reactissa siihen on useampia yhteisön tarjoamia moduleita, jotka toimivat eri tavoin ja joista käsiteltiin yhtä eli *React-authorization*-modulia. Sekä *React-authorization* että Angularin *CanActivateChild* -portti ovat samanlaisia sikäli, että molemmat vaikuttavat kaikkiin lapsikomponentteihin, ne estävät lapsikomponentteja näkymästä muille kuin tietylle roolille. Nämä kaksi eroavat toisistaan siten, että Angularin portit ovat reittikohtaisia, mutta *React-authorization:*n portit komponenttikohtaisia. Reactiin löytyy myös reittikohtaisia pääsynhallintamoduleita, kuten *react-router-role-authorization* ja Angulariin komponenttikohtaisia, kuten *ngsecurity* [React-router role authorization, 2017; Ngsecurity, 2015].

Kaikki kokeilemani pääsynhallintakirjastot sisältävät subjektikohtaisen auktorisoinnin. Resurssikohtaisuus löytyy joistakin kirjastoista, muttei kaikista. Resurssikohtaisen voi saada helposti käyttöön esimerkiksi antamalla metodille lisäparametriin.

Djangon, ASP.Netin ja Node.js-modulien pääsynhallintojen toimintaa käsitellään tarkemmin seuraavassa luvussa esimerkkiohjelmien kautta.

6 Esimerkkisovellukset

Tässä kappaleessa käydään läpi kolmea eri sovelluskehyksellä tehtyä esimerkkiohjelmaa, joiden pääsynhallintaa käsitellään tarkemmin käytännön toteutuksen avulla.

6.1 Node.js, Tietovarasto

Nodella toteutettu sovellus on tietovarasto, johon voi tallettaa tekstipohjaista dataa. Tietovaraston on tarkoitus sijaita pilvessä, sitä käytettäisiin verkon kautta ja pääsynhallinta liittyy datan käsittelyn sallimiseen ja estämiseen. Käyttäjillä tulee olla luvat CRUD-operaatioihin eli datan luomiseen, lukemiseen, muokkaamiseen ja poistamiseen.

Koska Node.js:ssä on useita moduleita, joita voidaan käyttää pääsynhallinnassa ja ohjelman muita osia voidaan helposti käyttää uudelleen, päätin toteuttaa Nodella useamman eri auktorisoinnin. Sovelluksessa on yksi tarve auktorisoinnille, josta on monta toteutusta, jotka sijaitsevat eri kansioissa ja ne on toteutettu eri moduleilla. Joka toteutuksella on oma tiedosto reititykselle, joka sisältää reititysfunktion jokaiselle CRUD-operaatiolle, joissa pääsynhallinta tapahtuu. Esimerkiksi relatiivinen osoite */delete/* ohjautuu omaan reititysfunktioonsa, jossa tarkastetaan, saako käyttäjä poistaa dataa.

Toteutukset käyttävät yhteisiä MongoDb-tietokantoja. Tietokanta datalle sijaitsee tiedostossa db.js, joka sisältää metodit jokaiselle CRUD-operaatiolle, joissa tiedon muokkaaminen, lukeminen, poistaminen tai luominen varsinaisesti tapahtuu tietokannassa. Näitä metodeita kutsutaan siis reititysfunktioista, kun käyttäjän pääsy on tarkistettu. Tietokanta käyttäjille on tiedostossa user.js, jossa käyttäjälle on tallennettu yksi attribuutti jokaiselle CRUD-operaatiolle totuusarvona sekä käyttäjän rooli merkkijonona. Attribuuttien, joita joka käyttäjällä on siis neljä, pienempi määrä olisi ollut helpommin hallittavissa.

Koko ohjelmalle yhteinen sessionhallinta on tehty *Express-session*-modulin avulla, jossa session tunnus tallennetaan evästeeseen ja eväste allekirjoitetaan satunnaisesti muodostetulla merkkijonolla. Evästeen voimassaoloajaksi on asetettu 20 minuuttia ja sillä on HttpOnly-lippu, jolloin sitä ei saada käyttäjän puolella näkyviin skriptillä. Session tieto tallennetaan välimuistiin ja sen asetuksissa on määritelty, ettei evästettä, jota ei ole muokattu pyynnössä, tallenneta turhaan uudestaan.

Aloitin oman autentikoinnin tekemisen, mikä toimisi Expressillä, mutta se osoittautui aikaavieväksi ja bugialttiiksi. Lopulta päädyin käyttämään *Passport*-modulia, jossa autentikointi on pääpiirteittäin toteutettu valmiiksi. *Passport*:n paikallinen autentikointistrategia etsii tietokannasta ensin käyttäjänimen ja jos käyttäjänimi löytyy, se tarkistaa täsmääkö salasana siihen. Autentikointi on siis kaksiosainen ja käyttää samaa koodia kuin alkuperäinen oma toteutus, mutta välttää alkuperäisen bugit. Jos tunnistautuminen epäonnistuu, uudelleenohjataan login-sivulle. Käyttäjä sarjallistetaan sessioon id:ksi eli käyttäjäoliosta tallennetaan sessioon pelkkä id ja kun käyttäjää tarvitaan pyyntöobjektissa, käyttäjän id rekonstruoidaan takaisin täydeksi käyttäjäobjektiksi.

Seuraavissa alakohdissa esitellään Expressin pääsynhallintamoduleita, niiden toimintaa ja kyseisen modulin toteutus tietovarastosovellukseen. Jokaisesta pääsynhallintamodulin toteutus on sijoitettu omaan kansioonsa. Kahdella ensimmäisellä modulilla toteutin attribuuttipohjaisen pääsynhallinnan.

6.1.1 Node-abac

Tietovarastossa määrittelin politiikan JSON-muodossa. Tässä tapauksessa koodiesimerkissä 3 subjektityyppejä on vain yksi, käyttäjä, jolle määritellään mahdollisiksi attribuuteiksi *readAttribute, createAttribute, updateAttribute ja removeAttribute*. Nämä vastaavat oikeuksia datan lukemiseen, luomiseen, muokkaamiseen ja poistamiseen. Koodiesimerkissä 3 sääntöihin (*rules*) on määritelty lukuoikeus (*can-read*), joka toteutuu, jos käyttäjällä on lukuattribuutti *readAttribute*. Sovelluksessa lukuattribuutin tyyppi on boolean, ja sen arvon tulee olla tosi (*true*) ja sen vertailuun käytetään ”&&”-operaattoria (*boolAnd*).

```
2  const policies = {
3    attributes:{
4      user:{
5        /* Subjektityypin mahdolliset attribuutit */
6        readAttribute: "Can read data",
7        removeAttribute: "Can delete data",
8        createAttribute: "Can create data",
9        updateAttribute: "Can update data"
10     }
11   },
12   rules:{
13     /* Säännön nimi */
14     "can-read":{
15       /* Sääntöön tarvittavat attribuutit */
16       attributes:{
17         "user.readAttribute":{
18           /* Odotettu arvo */
19           comparison_type: "boolean",
20           comparison: "boolAnd",
21           value: true
22         }
23       }
24     },
```

Koodiesimerkki 3. Node-Abacin sääntöjen määrittely.

Node-abac -modulissa pääsyoikeus tarkistetaan *enforce*-metodilla, jota käytetään koodiesimerkissä 4 rivillä 37, jossa sillä tarkistetaan, onko käyttäjällä lukuoikeus dataan. Sen ensimmäinen parametri *can-read* on lukuoikeuden nimi ja toinen parametri *current_user* sisäänkirjautunut käyttäjä. Jos käyttäjällä on lukuoikeus, *enforce* palauttaa arvon *true* ja data luetaan tietokannasta rivillä 38. Vastaavasti jos käyttäjällä ei ole lukuoikeutta, *enforce* palauttaa false, jolloin mennään riville 52, jossa kutsutaan koodiesimerkin ulkopuolella määriteltyä *unathorized*-funktiota, jonka tehtävä on lähettää virhevastaus statuskoodilla 403.

```
31  /*Abac otetaan käyttöön */
32  const Abac = new nodeAbac(policies);
33
34  /* reititys lukemiseen */
35  const read = app.get('/read/:id', function(req,res){
36
37    if( Abac.enforce('can-read', current_user) ){
38      db.readData(req.params.id, function(data){
39        if(data){
40          res.json(
41            {
42              "data": data
43            }
44          );
45        }
46        else{
47          not_found(req,res);
48        }
49      });
50    }
51    else{
52      unauthorized(err,res);
53    }
54  });
```

Koodiesimerkki 4. Funktio datan lukemisen reititykseen.

6.1.2 Abac

Käytin tietovarastossa *Abac*-modulin valmista *InMemory* -tallennustapaa, joka tallentaa politiikan JavaScript-objektina muistiin ja tietovarastossa se on asetettu NAME-vakioon. Lukuoikeussääntö määritellään koodiesimerkin 5 rivillä 54, jossa *setPolicy*-metodille on annettu ensimmäisenä parametrina kyseinen tallennustapa ja toisena parametrina säännön nimi ’*read*’. Kolmantena parametrina annettu funktio yksinkertaisesti tarkastaa onko käyttäjän readAttribute tosi ja palauttaa joko *true* tai *false*. Käyttäjä on tallennettu pyyntöobjektin kenttään user.user, jotta sekä sessionhallinta että salasanan hajautus toimivat teknisesti oikein.

Myös pääsyoikeuden tarkistavalle can-metodille annetaan säännön nimen lisäksi tallennustavan nimi. Koodiesimerkissä 5 *can*-metodia kutsutaan rivillä 65 ja rivillä 67 *yes*-funktio kutsuu funktiota *readData,* joka nimensä mukaisesti lukee datan tietokannasta sen jälkeen kun pääsy on sallittu. Vastaavasti pääsyn epäämisen jälkeen *no*-funktio kutsuu *unauthorized*-funktiota rivillä 70, joka lähettää ”HTTP Error 401 Unauthorized” -viestin.

Tietovarastosovelluksessa metodia *can* käytetään reititysmetodin sisällä, mutta metodin voi asettaa myös yhdeksi retitityksmetodin parametriksi. Tätä tapaa en käyttänyt tietovarastosovelluksessa, koska req.user on tyhjä silloin, kun käyttäjä ei ole kirjautunut sisään, mitä parametrissa sijaitseva *can* -metodi ei ota huomioon. Sisäänkirjautumattoman käyttäjän ongelma hoidetaan koodiesimerkissä 5 rivillä 61, jolloin jos pyynnössä ei ole käyttäjäobjektia, tehdään uudelleenohjaus sisäänkirjautumissivulle.

```
44  //Politiikat
45  abac.set_policy(NAME, 'delete', function(req){
46    return (req.user.user.deleteAttribute == true ? true : false);
47  });
48  abac.set_policy(NAME, 'update', function(req){
49    return (req.user.user.updateAttribute == true ? true : false);
50  });
51  abac.set_policy(NAME, 'create', function(req){
52    return (req.user.user.createAttribute == true ? true : false);
53  });
54  abac.set_policy(NAME, 'read', function(req){
55    return (req.user.user.readAttribute == true ? true : false);
56  });
57
58  /* reititys lukemiseen */
59  const read = app.get('/', function(req, res, next){
60
61    if(!req.user){
62      return res.redirect('/login');
63    }
64
65    abac.can(NAME, 'read', {
66      yes: function() {
67        readData(req,res);
68      },
69      no: function(err, info) {
70        unauthorized(req,res);
71      }
72    })(req, res);
73  });
```

Koodiesimerkki 5. Abac-modulin poltiikka ja lukureititys.

Moduleilla Node-abac ja Abac toteutin siis attribuuttipohjaisen pääsynhallinnan. Kolmannella Accesscontrol-modulilla tein roolipohjaisen auktorisoinnin, jotka varten lisäsin käyttäjälle tietokantaan roolisarakkeen.

6.1.3 Accesscontrol

Accesscontrol-modulissa sääntöjä luodaan *grant*-metodilla, jolle annetaan roolin nimi ja sen jälkeen ketjutetaan oikeuksia tiettyyn objektiin. Koodiesimerkin 6 rivillä 32 määritel-

lään käyttäjärooli *admin*, jolle annetaan readAny-oikeus sekä updateAny-oikeus objektille *'data'*. En käyttänyt lainkaan *Own*-päätteisiä oikeuksia subjektin omistamaan objektiin. Objektit ovat käytännössä merkkijonoja.

Koodiesimerkissä käytetään *can*-metodia rivillä 42, jossa otetaan pääsyoikeusolio peruskäyttäjän roolille *user* ja readAny-oikeudelle. Rivillä 44 tarkistetaan totuusarvo if-lauseessa, joka saadaan pääsyoikeusolion *granted*-metodista. Käyttäjäoliossa rooli on siis tekstinä.

```
26  const ac = new AccessControl();
27
28  // Käyttäjärooli
29  ac.grant('user')
30    .readAny('data')
31  // Admin
32  .grant('admin')
33    .readAny('data')
34    .updateAny('data');
35
36  /* Reititys lukemiseen */
37  const read = app.get('/', function(req, res, next){
38    if(!req.user){
39      return res.redirect('/login');
40    }
41
42    const permission = ac.can(req.user.user.role).readAny('data');
43
44    if(permission.granted){
45      readData(req,res);
46    }
47    else{
48      unauthorized(req,res);
49    }
50  });
```

Koodiesimerkki 6. Accesscontrol-moduli.

6.2 Django, Kurssisovellus

Djangolla toteutettu esimerkkisovellus on kurssisovellus, jonka tarkoitus on esittää miten auktorisointi, on toteutettu Djangolla ja sen liitännäisellä.

Kurssisovelluksen osat:

- sivu, jossa on luettelo kaikista kursseista,
- yksittäisen kurssin ja sen arvosanan sivu,
- sisäänkirjautumissivu,
- ja jokaisessa sivussa on päävalikko.

Päätin toteuttaa Djangolla roolipohjaisen pääsynhallinnan, koska roolipohjaisuus on Djangossa pitkälle sisäänrakennettuna. Sovelluksen sisäänkirjautumisessa ja käyttäjän tunnistautumisessa käytetään Djangon oletuksena toteuttamaa laajaa käyttäjänhallintaa.

Roolipohjaisuus sopii kurssisovellukseen, koska siinä on kolme roolia, joilla on selkeät tehtävät: opiskelija, opettaja ja *admin*. Opiskelija voi ilmoittautua kurssille, jolle opettaja antaa arvosanan ja *admin* saa lisätä uuden kurssin. Kurssin malliobjektissa on kurssin nimi sekä listat opiskelijoista ja opettajista. Arvosanan malliobjektissa on itsensä arvosanan tieto mille kurssille ja kelle opiskelijalle se on annettu.

Pääsynhallinnan konfiguroimiseen meni paljon aikaa ja koska pääsynhallinnan toteuttaminen useammalla liitännäisellä vaatisi vielä enemmän muutoksia Djangon omiin asetuksiin, päätin käyttää liitännäisistä pelkästään Django Guardiania. Lopulta suurin osa oli auktorisoinnista toteutettu Djangon sisäänrakennetulla pääsynhallinnalla, jota esitellään seuraavassa alakohdassa.

6.2.1 Djangon sisäänrakennettu pääsynhallinta

Kurssisovelluksen päävalikko sisältää linkit eri toiminnallisuuksiin. Linkkien sivuille, joihin käyttäjän roolilla ei ole pääsyä, näkyminen estetään päävalikossa, jos käyttäjällä ole kyseistä oikeutta. Kaikki roolin oikeudet periytyvät käyttäjän oikeuksiksi, jos käyttäjä kuuluu rooliin. Koodiesimerkin 7 rivillä 25 otetaan käyttäjän oikeudet *auth*-modulissa (*perms.auth*) muuttujaan *permissions*. Rivillä 27 tarkistetaan kuuluuko kurssien katseluoikeus (*authorization.view_courses*) käyttäjän oikeuksiin.

```
15  <ul id="nav">
16
17    <li> <a href="{% url 'index' %}">Index</a> </li>
18
19    {% if user.is_authenticated %}
20      <li> <a href="{% url 'logout' %}">Log out</a> </li>
21    {% else %}
22      <li> <a href="{% url 'login' %}">Log in</a> </li>
23    {% endif %}
24
25    {% with permissions=perms.auth %}
26
27      {% if 'authorization.view_courses' in permissions %}
28        <li> <a href="{% url 'list' %}"> List courses </a> </li>
29      {% endif %}
30
31      {% if 'authorization.add_courses' in permissions %}
32        <li> <a href="{% url 'add_course' %}"> Add course </a></li>
33      {% endif %}
34
35    {% endwith %}
36
37  </ul>
```

Koodiesimerkki 7. Kurssisovelluksen päävalikon malline.

Koodiesimerkissä 8 on Python-funktio näkymälle, joka listaa kurssit. Auktorisoinnin kannalta keskeinen on koodirivi 23, jossa tarkistetaan, että nykyisellä käyttäjällä on oikeus kurssien katsomiseen funktiolla *has_perm*. Jos käyttäjä saa katsoa kursseja, haetaan kurssiobjektit, jotka muutetaan rivillä 28 sanakirjoja sisältäväksi listaksi. Seuraavalla rivillä palautetaan malline "list.html", jolle annetaan kurssit *context*-sanakirjassa. Jos käyttäjällä ei ole oikeutta kurssien katsomiseen, palautetaan 403-virhevastaus, mikä tapahtuu koodirivillä 30.

```
22  def list(request):
23      if request.user.has_perm('authorization.view_courses'):
24          courses = Course.objects.values('name','id')
25          course_list = []
26          if courses:
27              # to list of dicts
28              course_list = [entry for entry in courses]
29          return render(request,'list.html',context={'courses': course_list})
30      else:
31          return HttpResponseForbidden('<h1>Forbidden 403</h1>')
```

Koodiesimerkki 8. Kurssinlistausnäkymä.

Oikeus kurssien katsomiseen periytyy käyttäjäryhmältä yksittäiselle käyttäjälle. Tämän lisäksi Djangossa voi asettaa käyttäjälle yksittäisiä oikeuksia. Kurssisovelluksessa kaikki pääsyoikeudet on annettu ryhmän tasolla, vaikka Django mahdollistaisi myös yksittäiselle käyttäjälle annetut oikeudet.

6.2.2 Django Guardian

Kurssisovelluksen auktorisoinnissa on käytetty sisäänrakennetun pääsynhallinnan lisäksi Django Guardian -liitännäistä. Django Guardian (www.djangoguardian.readthedocs.io/) tarjoaa resurssikohtaisen auktorisoinnin, joka on yhdistetty lisäosattoman Djangon ominaisuuksiin. Guardian lisää resurssikohtaisuuden myös ylläpidon hallintasivulle ja tyhjälle käyttäjälle. Django Guardian otetaan käyttöön lisäämällä se konfiguraation INSTALLED_APPS-listaan sekä sen palvelinosa AUTHENTICATION_BACKENDS-listaan [Balcerzak, 2017].

Django Guardian lisää uudet versiot Djangon oman pääsynhallinnan mallineiden ja näkymien funktioista, joille annetaan lisäparametrina malliobjekti. Guardianin versiolle *has_perm*-metodista annetaan oikeuden nimen lisäksi malliobjekti, johon oikeus liittyy. Oikeus lisätään *assign_perm*-metodilla, jolle annetaan oikeuden nimi, käyttäjä ja malliobjekti. Myös *@permissions_required* -koristajalle on Django Guardianissa objektikohtaiset versiot [Balcerzak, 2017].

Kurssisovelluksessa Django Guardiania käytetään yksittäisen kurssin mallineessa, joka on kuvattu koodiesimerkissä 9. Siinä otetaan käyttäjän (*request.user*) oikeudet kurssiin (*course*) muuttujaan *course_perms* ja tarkistetaan seuraavalla rivillä, sisältyykö kyseiseen kurssiin opetusoikeus ('*teach_course*') muuttujaan. Jos kirjautunut käyttäjä on

yksi kurssin opettaja, opetusoikeus sisältyy *course_perms*:n ja malline näyttää lomakkeen, jolla arvosanan voi antaa. Muutoin näytetään kurssin arvosana, joka on saatu näkymästä *context*-sanakirjassa. Malline siis muuttaa muotoaan käyttäjäroolin ja malliobjektin mukaan.

```
7       {% get_obj_perms request.user for course as "course_perms" %}
8       {% if 'teach_course' in course_perms %}
```

Koodiesimerkki 9. Yksittäisen kurssin malline.

6.3 ASP.Net, QAEngine

C#:lla .NET Core -kehyksellä tehty sovellus on nimeltään QAEngine, joka mallintaa Quoran ja Stack Overflow:n kaltaista palstaa, jossa voidaan kysyä kysymyksiä ja vastata niihin. Kysymykset ja vastaukset muodostavat ketjuja. Sisäänkirjautumisen jälkeen käyttäjä voi kysyä kysymyksen, joka luo uuden ketjun ollen samalla ketjun aloitusviesti. Muut käyttäjät voivat vastata kysymykseen vastausviesteillä ja kysyjä valitsee niistä parhaan vastauksen.

Pääsynhallintaa tarvitaan tarkistamaan, onko käyttäjä kirjautunut sisään ja jotta käyttäjä saisi muokata vain omia kysymyksiään tai vastauksiaan. Ensiksi mainittuun voidaan käyttää parametritonta Authorize-koristajaa ja jälkimmäiseen tulee toteuttaa resurssikohtaista pääsynhallinta. Roolit ja vaatimukset (eli attribuutit) eivät sovi QAEnginen tapaukseen. Auktorisointi otetaan käyttöön tuomalla tiedostoon *Microsoft.AspNetCore.Authorization* -pakkaus, jolloin sen luokat saadaan käyttöön tiedostossa.

```
26          // GET:Question
27          [Authorize]
28          public ActionResult Index(int id)
29          {
30              using (ApplicationDbContext db = new ApplicationDbContext())
31              {
32                  var question = db.Questions.FirstOrDefault( q => q.Id == id);
33                  ViewData["Question"] = question;
34
35                  var answers = db.Answers.
36                          Where(a => a.QuestionId == id)
37                          .ToList();
38
39                  ViewData["Answers"] = answers;
40
41
42                  return View();
43              }
44
45          }
```

Koodiesimerkki 10. Kysymyksen näyttäminen.

QAEngine:ssä on luotu mallit kysymyksille (*QuestionModel*), vastauksille (*AnswerModel*) ja käyttäjälle (*ApplicationUser)*, joista viimeksi mainittu perii ASP.Net:n oletuskäyttäjäluokan *IdentityUser*. Mallit keskustelevat suoraan tietokantataulujen kanssa. Kysymyksen mallin kenttiä ovat id, päivämäärä, kirjoittajan id, otsikko ja teksti, jotka vastaavat tietokannan sarakkeita. Kun käyttäjä luo uuden kysymyksen, hän kirjoittaa siihen otsikon ja tekstin ja kontrolleri luo muut kentät automaattisesti nykyisen ajanhetken ja käyttäjänimen perusteella.

Kysymyksillä on oma kontrolleri *QuestionController*, joka sisältää metodit kysymyksen luonnille ja editoinnille. Koodiesimerkissä 10 on kontrollerimetodi kysymyksen näyttämiselle, jolle annetaan parametrina kysymyksen id, joka saadaan url-osoitteen (/Question/Index/1) polun lopusta numerona. Rivillä 32 haetaan tietokannasta se kysymys, jolla on kyseinen id ja kysymys asetetaan seuraavalla rivillä näkymän datasanakirjaan ViewData, josta se saadaan käyttöön näkymässä.

Koodiesimerkissä 10 pääsynhallintaa on rivin 27 Authorize-attribuutti. Tässä tapauksessa attributille ei ole annettu parametreina rooleja eikä vaatimuksia, jolloin pelkästään tarkistetaan onko käyttäjä kirjautunut sisään. Kun attribuutti on asetettu paikoilleen, Asp.net päästää vain sisäänkirjautuneen käyttäjän Index-metodiin. Jos sisäänkirjautumaton käyttäjä pyrkii metodin osoitteeseen, hänet ohjataan sisäänkirjautumissivulle.

```
44          // This method gets called by the runtime. Use this method to add services
45          public void ConfigureServices(IServiceCollection services)
46          {
47              // Add framework services.
48              services.AddDbContext<ApplicationDbContext>(options =>
49                  options.UseMySQL(Configuration.GetConnectionString("mysqlconn")));
50
51              services.AddIdentity<Models.ApplicationUser, IdentityRole>()
52                  .AddEntityFrameworkStores<ApplicationDbContext>()
53                  .AddDefaultTokenProviders();
54
55              services.AddMvc();
56
57              services.AddAuthorization( options => {
58                  options.AddPolicy("IsAuthorPolicy", policy =>
59                      policy.Requirements.Add(new IsAuthorRequirement())
60                  );
61              });
62
63              services.AddSingleton<IAuthorizationHandler, IsAuthorHandler>();
```

Koodiesimerkki 11. Auktorisoinnin konfigurointi.

Resurssikohtaiseen pääsynhallintaan vaaditaan enemmän toimia kuin yksinkertaisien koristajien lisääminen: konfiguraatioon tulee lisätä kaikki pääsynhallintasäännöt (politiikat) ja niiden kriteerit (vaatimukset) luokkina. Koodiesimerkissä 11 rivillä 57 palvelukokoelmaan lisätään auktorisointipalvelu *AddAuthorization*-metodilla. Auktorisointipalveluun lisätään politiikka nimeltä *IsAuthorPolicy*, joka on sääntö tekijän tarkistamiselle ja

jolle on asetettu yksi kriteeri *IsAuthorRequirement.* Kriteeri *IsAuthorRequirement* on oma luokkansa, joka vaatii, että kysymyksen muokkaajan tulee olla sama kuin sen tekijä, mutei sisällä QAEnginen tapauksessa mitään dataa tai metodeita. On mahdollista rakentaa luokkarakenne toisella tavalla siten, että kriteeri sisältäisi resurssin datan, esimerkiksi kysymyksen tiedot ja tekijän nimen, josta käsittelijäluokka saisi ne käyttöönsä. Kriteeriluokkaa tarvitaan tässä tapauksessa vain käsittelijäluokalle, jotta se voisi merkitä kyseisen kriteerin täyttyneeksi.

```
11  public class IsAuthorHandler :
12                      AuthorizationHandler<IsAuthorRequirement, QuestionModel>
13  {
14      private ApplicationDbContext db = new ApplicationDbContext();
15
16      protected override Task HandleRequirementAsync(
17          AuthorizationHandlerContext context,
18          IsAuthorRequirement requirement,
19          QuestionModel question
20      )
21      {
22          var poster = db.Users.FirstOrDefault(u => u.Id == question.Username);
23
24          question.Poster = poster;
25
26          if (context.User.Identity?.Name == question.Poster.UserName)
27          {
28              context.Succeed(requirement);
29              return Task.FromResult(true);
30          }
31          return Task.FromResult(false);
32
33      }
34  }
```

Koodiesimerkki 12. Kysymyksen tekijän tarkistava käsittelijäluokka.

Varsinaisesti *IsAuthorHandler*-käsittelijäluokka tekee tarkistuksen, onko kysymyksen kirjoittanut käyttäjä sama kuin sisäänkirjautunut käyttäjä. Koodiesimerkissä 12 tämä tapahtuu luokan *HandleRequirementAsync*-metodissa rivillä 26, jossa sisäänkirjautunut käyttäjä saadaan kontekstin *User.Identity.Name*-kentästä ja kysymyksen tehneen käyttäjän nimi kysymysolion lähettäjän käyttäjänimestä *question.Poster.UserName*. Jos kirjautuneen käyttäjän nimi on sama kuin kysymyksen kirjoittajan nimi, asetetaan rivillä 28 kriteeri täyttyneeksi kontekstiolion *Succeed*-metodilla. Tietokannan kontekstiolio *context* sisältää nykyisen käyttäjän, kaikki säännön kriteerit ja tiedon siitä, ovatko kriteerit täyttyneet. *HandleRequirementAsync* saa parametreina kontekstin lisäksi kysymyksen ja kriteeriluokan *IsAuthorRequirement* oliot. Asynkronisena metodina se palauttaa *Task*-olion, joka paketoi sisäänsä totuusarvon.

Edellisessä koodiesimerkissä 11 rivillä 63 sovelluksen palveluihin asetetaan *IsAuthorHandler*-luokka toteuttamaan auktorisointikäsittelijäpalvelu *IAuthorizationHandler* riippuvuusinjektiolla. *AddSingleton* metodi lisää riippuvuuden *singleton*-tyyppisenä, jolloin käsittelijäpalvelusta on vain yksi globaali ilmentymä [Microsoft .Net Docs 2017].

```
96          // GET: Question/Edit/5
97          public async Task<ActionResult> Edit(int id)
98          {
99              using( ApplicationDbContext context = new ApplicationDbContext())
100             {
101
102                 var question = context.Questions.FirstOrDefault();
103
104                 var user = User;
105
106                 bool authorizationSuccess = await authorizationService.AuthorizeAsync(user, question, "IsAuthorPolicy");
107
108
109                 if (authorizationSuccess)
110                 {
111                     return View();
112                 }
113                 else
114                 {
115                     return Unauthorized();
116                 }
117
```

Koodiesimerkki 13. Kysymyksen editointi.

Koodiesimerkissä 13 on kysymyksen kontrollerin muokkausmetodi, jolle annetaan parametrina muokattavan kysymyksen id, jolla haetaan kysymys tietokannasta rivillä 102. Kun politiikka *IsAuthorPolicy* sekä luokat kriteerille ja sen käsittelijälle on määritelty, voidaan politiikalle kutsua rivillä 106 *AuthorizeAsync*-metodia, jolle annetaan käyttäjä, kysymys ja politiikan nimi. *AuthorizeAsync*:n palauttama totuusarvo asetetaan muuttujaan *authorizationSuccess*, jonka perusteella päätetään annetaanko käyttäjälle muokkausnäkymä View()-metodilla vai Unauthorized (410) -vastaus. Muokkausnäkymä sisältää lomakkeen, jolla käyttäjä voi antaa uudet arvot kysymykselle. Kun käyttäjä palauttaa lomakkeen, kutsutaan muokkausmetodin POST-versiota.

6.4 Esimerkkisovellusten yhteenveto

Tässä kohdassa vertaillaan keskenään Express, Django ja Asp.Net -kehysten auktorisointia, joilla toteutin esimerkkiohjelmat.

Käsittelin kahta melko samannimistä attribuuttipohjaista Node.js-modulia, *Abac* ja *Node-abac*. Ensiksi mainitussa politiikan luominen *setPolicy* -metodia kutsumalla on helpompaa kuin *Node-abac*-modulissa, jossa täytyy luoda monimutkainen JSON -objekti, mutta toisaalta sen politiikka voi olla monipuolinen. *Abac*-modulin huonona puolena tallennustavan nimeä joutuu käyttämään joka paikassa ”ylimääräisenä” parametrina, myös pääsyä tarkistettaessa, mikä tuo turhaa monimutkaisuutta ja tallennustavan nimeä voi joutua siirtämään tiedostosta toiseen. *Node-abac*:ssa tallennustavan nimestä ei tarvitse huolehtia ja pääsyn tarkistus toimii myös resurssikohtaisesti, jolloin pääsyn tarkistus on toteutettu paremmin kuin *Abac*:ssa.

Abac-modulin dokumentaatio on puutteellinen, vain peruskäyttö kuvataan ja dokumentaatiossa ei selitetä tallennustapaa eikä lisäasetuksia. Node-abacin dokumentaatio on kattavampi, koska se sisältää enemmän esimerkkejä ja lisätietoa attribuuttien vertailusta.

Accesscontrol poikkeaa muista siten, että oikeudet on rajattu CRUD-operaatioihin joko omille tai kaikille resursseille. Lisäksi se on muista poiketen roolipohjainen eikä attribuuttipohjainen. Mahdollisia oikeuksia, joita voidaan asettaa rooleille, on siis annettu valmiiksi kahdeksan, mutta roolit voidaan määritellä itse. Myös Djangossa on toteutettu valmiiksi käyttäjän CRUD-oikeudet, mutta pääsynhallinnan käyttö ei ole sidottua vain niihin: CRUD-oikeuksia ei tarvitse käyttää ollenkaan ja omia sääntöjä saa lisätä.

Acce*sscontrol*:ssa oikeuksien antaminen *grant*-operaattorilla on kätevää, koska sillä määritellään samalla sekä oikeudet että roolit ja moduli osaa tilanteen mukaan luoda uuden roolin tai päivittää vanhan oikeuksia. Oikeus annetaan aina jollekin resurssille, vaikka pääsy ei olisi resurssikohtaista, missä tapauksessa pitää keksiä jokin yleinen nimi. Tämän ajatustavan omaksuminen vei aikaa. Tietovarasto-sovelluksessa resurssin nimi on ”data”.

Asp.Netin auktorisointi on työläs konfiguroida riippuvuusinjektiolla. Politiikan käsittelijän rekisteröimiseen on useita metodeita, joista pitää huomata, että ainoastaan käsittelijän *singleton* -tyyppisenä rekisteröivä toimii oikein. Asp.Net:ssä tehdään politiikalle, kriteerille ja käsittelijälle jokaiselle oma luokka ja käsittelijäluokassa joutuu toteuttamaan käyttäjän ja mallin hakemisen itse. Kielenä C# on vahvasti oliopohjainen ja Asp.Net ”raskaampi” kehys kuin esimerkiksi Express, joten työläys johtuu ohjelmointikielen ja kehyksen ominaisuuksista eikä pelkästään auktorisoinnin toiminnasta.

Djangon auktorisoinnin käyttöönotto vaati jonkin verran vaivaa, mutta vähemmän kuin Asp.Net:ssä. En käyttänyt Djangon koristajia, mutta reititysmetodeissa olisi voinut käyttää *permission_required()*-koristajaa, joka olisi ajanut saman asian kuin *has_perm()*-metodit. Koristajalla reititysmetodin sisältö olisi ollut itse asiassa yksinkertaisempi, koska sillä olisi välttänyt *if-else* -rakenteet. Toisin kuin Djangossa, Asp.Net:ssä käytin koristajia, sillä koristajat ovat siinä pääasiallinen tapa oikeuksien tarkistukseen.

Taulukoissa 1 ja 2 vertaillaan esimerkkisovelluksissa käytettyjä pääsynhallintoja.

	Node-abac	**Abac**	**Access-control**	**Django**	**Asp.Net**
Tyyppi	Attribuutti	Attribuutti	Rooli	Rooli ja attribuutti	Rooli ja attribuutti
Oikeuksien luonti	Monimutkainen JSON tai Yaml-objekti, jossa on attribuutit ja säännöt.	Jokaiselle oikeudelle *set_policy* -metodikutsu.	Ketjutetut funktiokutsut: *grant* roolille ja oikeuden nimet oikeuksille. Luo roolin, jos sitä ei ole. Oikeudet voi asettaa objektille per kenttä.	*Add* -metodikutsut käyttäjälle tai ryhmälle. Manuaalisesti admin-näkymän kautta.	Riippuvuus-injektio. *AddPolicy*-kutsut joka oikeudelle.
Oikeuksien testauksen konfigurointi	Attribuuttien vertailua voi kustomoida oikeusobjektissa. Ei omia metodeita.	Oma tarkistusmetodi toteutetaan aina *setpolicy*:lle vastakutsufunktiona.	Ei omia metodeita. Resurssin omistuksen tarkistus tulee tehdä itse.	Voi kuormittaa testimetodin.	Resurssikohtaisessa tavassa oikeuksien testausta voi konfiguroida vapaasti.
Oikeuksien testaus	*Can* -metodi, jolla on *yes* ja *no* vastakutsufunktiot. Tallennustavan nimi kulkee turhaan mukana.	Enforce-metodi.	*Can*-metodi palauttaa olion, jonka *granted*-metodista saa totuusarvon.	*Has_perm*-metodilla. Mallineissa *permissions*-objektissa. Koristajia.	Koristajilla metodien tai luokkien yläpuolella.

Taulukko 1. Pääsynhallintojen oikeuksien vertailua.

	Node-abac	**Abac**	**Access-control**	**Django**	**Asp.Net**
Resurssi-kohtaisuus	Ei ole.	Kyllä, helppo siirtyä siihen, koska enforce-metodille annetaan vain lisä -parametri.	Aina resurssikohtaista, resurssi teknisesti merkkijono.	Ei sisään-rakennettuna, Django Guardian sisältää.	Kyllä.
Sääntöjen monipuoli-suus	Monipuoliset sääntöobjektit. Oikeus voi-daan asettaa riippumaan useammasta attribuutista. Vertailua voi kustomoida.	Yksipuolinen: yksi attribuutti per oikeus.	Kahdeksan valmiiksi nimettyä sääntöä CRUD-operaatiolle. Voi jakaa vapaasti roo-leille ja tehdä helposti uusia rooleja.	Subjekti ja objek-tipohjainen. Voi luoda omia omia sääntöjä. Oikeudet sekä ryhmillä, että yksittäisillä käyt-täjillä.	Monipuolinen. Si-sältää resurssi-kohtaisen ja atti-buuttipohjaisen pääsynhallinnan sekä politiikat. Resurssikohtaista voi kustomoida paljon.
Dokumen-taatio	Sisältää katta-vat esimerkit ja tarkennusta attribuuttien vertailuun.	Puutteellinen: ly-hyet esimerkit riit-täviä peruskäyt-töön, muttei seli-tetä tallennusta-poja eikä asetuk-sia.	Kattava dokumentaa-tio: paljon esimerk-kejä.	Samassa yhtey-dessä autentikoin-nin kanssa. Kat-tava dokumentaa-tio.	Kattava doku-mentaatio.
Käytettä-vyys ja käyttöön-otto	Oikeuksien luonti vaikeaa monimutkai-sella sääntöob-jektilla, mutta oikeuksien testaaminen sujuvaa.	Sekä sääntöjen luonti, että oi-keuksien tarkistus helppoa.	Ketjutettujen funk-tiokutsujen kanssa al-kuunpääsy hankalaa, mutta loppujen lopuksi sujuva käyttää.	Kohtuullisen helppo käyttää. Konfigurointiin meni jonkin ver-ran aikaa.	Konfigurointi ja erilaisten luok-kien luonti työ-lästä.

Taulukko 2. Pääsynhallintojen vertailua.

7 Tutkielman yhteenveto

Tämän tutkielman alussa määriteltiin pääsynhallinta ja käsiteltiin sitä teoreettisesta näkökulmasta. Sen jälkeen määriteltiin sovelluskehykset ja web-sovelluskehykset. Luvussa 5 esiteltiin eri web-sovelluskehysten pääsynhallintaa niiden dokumentaatioiden perusteella. Loppuosaan toteutettiin kolmella eri web-sovelluskehyksellä oma esimerkkiohjelma, joiden avulla kyseisten kehysten pääsynhallintaa tutkittiin tarkemmin.

Tutkielman jokainen esimerkkiohjelma oli erilainen. Eri kehyksillä olisi kuitenkin voinut toteuttaa saman esimerkkiohjelman, jolloin kehysten pääsynhallintaa olisi voinut vertailla suoraan. Yhdessä ohjelmassa olisi ollut määriteltynä tietyt kohdat, jossa pääsynhallintaa käytetään, joita vertailemalla kehysten välisten pääsynhallinnan tarkastelu olisi ollut helppoa. Esimerkkiohjelmat kuitenkin auttoivat käsittelemään kehysten pääsynhallintaa tarkemmin. Olisi ollut mielenkiintoista toteuttaa pääsynhallintaa muillakin kehyksillä, mutta siihen ei tutkielman puitteissa ollut aikaa.

Pääsynhallinnan keskeisimpinä tyyppeinä rooli- ja attribuuttipohjaisuus löytyvät kaikista kehyksistä. Resurssikohtainen auktorisointi löytyy useimmista kirjastoista ja sen saa helpostikin käyttöön esimerkiksi antamalla resurssiobjekti metodin lisäparametriksi. Koodissa pääsyoikeuden tarkistaminen perustuu totuusarvojen vertailuun ehtolauseissa tai joissain kehyksissä metodien yllä sijaitsevin koristajiin. Mallineissa oikeus tarkastetaan usein html-tageja muistuttavilla kontrollirakenteilla ja saatua totuusarvoa käytetään käyttöliittymäelementtien piilottamiseen. Sääntöjen monipuolisuus ja sijainti vaihtelee kirjastoittain. Osissa kirjastoista erilaiset säännöt on tarkasti rajattu, joissain taas omia sääntöä voi luoda vapaasti, mutta niiden tekeminen on turhan monimutkaista. Säännöt ja oikeudet voidaan luoda olioon tai esimerkiksi JSON-objektiin.

8 Viiteluettelo

Abac. (2014). Attribute based access control for node.js. Retrieved from https://github.com/vovantics/abac

Accesscontrol. (2018). Role and Attribute based Access Control for Node.js. Retrieved from https://www.npmjs.com/package/accesscontrol

Angular Architecture Docs. (2018). Retrieved from https://angular.io/guide/architecture

Angular Router Docs. (2018). Angular router guards. Retrieved from https://angular.io/guide/router#milestone-5-route-guards

Angular Typescript Docs. (2018). Retrieved from https://angular.io/guide/typescript-configuration

Alberts, C., & Dorofee, A. (2004). Security incident response: Rethinking risk management. Paper presented at the *International Congress Series.* 1268*,* 141-146.

Amuthan, G., & Aniket, S. (2014). Spring MVC beginner's guide. Packt Publishing.

Anderson, R., Boot, B., Latham, L., Addie, S., & Pasic, A. Authorization in ASP.NET c re. Retrieved from https://docs.microsoft.com/en-us/aspnet/core/security/authorization/

Bagnall, B., Faircloth, J., & Cabrera, H. (2002). C# for java programmers. Rockland, MA: Syngress.

Balcerzak, L. (2017). Django-guardian. Retrieved from https://django-guardian.readthedocs.io/en/stable/

Belosa, I. (2015). Understanding access control lists (ACL). Retrieved from http://www.routerfreak.com/understanding-access-control-lists-acl/

Belshe, M., Peon, R. & Thomson, M. (2015). Hypertext transfer protocol version 2 (HTTP/2). Retrieved from http://httpwg.org/specs/rfc7540.html

Butterfield, A., & Ngondi, G. (2016). A dictionary of computer science (7th ed.) Oxford University Press.

Cantelon, M., Harter, M., Holowaychuk, T. J., & Rajlich, N. (2014). Node.js in action Manning.

Colantonio, A., Di Pietro, R., & Ocello, A. (2012). Role mining in business: Taming role-based access control administration. Singapore: World Scientific.

Coyne, E. J., & Davis, J. M. (2008). Role engineering for enterprise security management. Boston: Artech House, Inc.

Django Software Foundation. (2018). Django documentation. Retrieved from https://docs.djangoproject.com/en/2.0/

Facebook. (2018). Jakamiesi asioiden näkyvyyden hallinta. Retrieved from https://fi-fi.facebook.com/help/1297502253597210/?helpref=hc_fnav

Fayad, M., & Schmidt, D., C. (1997). Object-oriented application frameworks.40(10), 32-38.

Ferraiolo, D. F., & Kuhn, D. R. (1992). Role-based access controls.

Ferraiolo, D., Chandramouli, R., & Kuhn, D. R. (2007). Role-based access control (2nd ed.). Boston: Artech House.

Fielding, R. T., & Taylor, R. N. (2000). Architectural styles and the design of network-based software architectures University of California, Irvine Doctoral dissertation.

Fielding, R., Gettys, J., Mogul, J., Nielsen, H., Masinter, L., Leach, P. & Berners-Lee, T. (1999). Hypertext transfer protocol -- HTTP/1.1. Retrieved from https://www.w3.org/Protocols/rfc2616/rfc2616.html

Fowler, M. (2015) Inversion of control. Retrieved from https://martin-fowler.com/bliki/InversionOfControl.html

Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., & Stewart, L. (1999). HTTP authentication: Basic and digest access authentication. RFC 2617.

Gegick, M., & Barnum, S. (2013). Least common mechanism. Retrieved from https://www.us-cert.gov/bsi/articles/knowledge/principles/least-common-mecha-nism

Harrington, J. L. (2009). Relational database design and implementation: Clearly explained (3rd ed.). Amsterdam; Boston: Morgan Kaufmann/Elsevier.

Herron, D. (2013). Node web development (2nd; 2 ed.). Birmingham: Packt Publishing.

Hot Frameworks. (2018). Retrieved from https://hotframeworks.com/

Hu, V. C., Ferraiolo, D., Kuhn, R., Friedman, A. R., Lang, A. J., Cogdell, M., Schnitzer A., Sandlin, K., Miller, R., & Scarfone, K. (2014). Guide to attribute based access control (ABAC) definition and considerations. NIST Special Publication, 800(162)

Ince, D. (2013). A dictionary of the internet (3rd ed.)

Jones, M., Bradley, J., & Sakimura, N. (2015). JSON web token (JWT). Retrieved from https://www.rfc-editor.org/rfc/rfc7519.txt

Kerrisk, M. (2010). The linux programming interface: A linux and UNIX system programming handbook (1st ed.). San Francisco: No Starch Press.

Kothari, D. P. (2011). Linux. Daryaganj, New Delhi, India: New Age International (P) Ltd., Publishers.

Krajka, B. (2015). The difference between virtual DOM and DOM. Retrieved from http://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/

Kuhn, D. R., Coyne, E. J., & Weil, T. R. (2010). Adding attributes to role-based access control. Computer, 43(6), 79-81. 10.1109/MC.2010.155

Lampson, B. (1974). Protection. ACM SIGOPS Operating Systems Review, 8(1), 18-24. 10.1145/775265.775268

Latham, D. C. (1986). Department of defense trusted computer system evaluation criteria. Department of Defense.

Lewis, A. (2014). Rails crash course: A no-nonsense guide to rails development. San Francisco: No Starch Press. Retrieved from http://ebookcentral.proquest.com/lib/tampere/detail.action?docID=1842153

Lipner, S. B. (2015). The birth and death of the orange book. *IEEE Annals of the History of Computing, 37*(2), 19-31. 10.1109/MAHC.2015.27

Makai, M. (2017). Web frameworks. Retrieved from https://www.fullstackpython.com/web-frameworks.html

McCool, S. (2012). *Laravel starter*. Birmingham: Packt Publishing.

Microsoft. (2017). .Net dependency injection API. Retrieved from
https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.servicecollectionserviceextensions.addsingleton

Morgan, T. (2010). HTTP digest integrity.

Mozilla Foundation. (2017) 1. Callback function. Retrieved from https://developer.mozilla.org/en-US/docs/Glossary/Callback_function

Mozilla Foundation. (2017) 2. Cross-origin resource sharing. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Mozilla Foundation. (2017) 3. JavaScript guide. Retrieved from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

Mozilla, Foundation. (2017) 4. Same origin policy. Retrieved from https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

Mrdovic, S., & Perunicic, B. (2008). Kerckhoffs' principle for intrusion detection. Paper presented at the *Telecommunications Network Strategy and Planning Symposium, 2008. Networks 2008. the 13th International,* 1-8.

Nemeth, G. (2015). Web authentication methods explained. Retrieved from https://blog.risingstack.com/web-authentication-methods-explained/

Ngsecurity. (2016). Retrieved from https://www.npmjs.com/package/ngsecurity

Nikiforakis, N., Meert, W., Younan, Y., Johns, M., & Joosen, W. (2011). SessionShield: Lightweight protection against session hijacking. Paper presented at the *International Symposium on Engineering Secure Software and Systems,* 87-100.

Node-Abac. (2015). Simon Barton. Node.js attributes based access control library. Retrieved from https://github.com/simon-barton/node-abac

Oracle. (2014). Java documentation. Retrieved from https://docs.oracle.com/javase/8/docs/

Otwell, T. (2017). Laravel docs. Retrieved from https://laravel.com/docs/5.6/authorization

Parecki, A. (2018). *OAuth 2.0 simplified;* (2nd ed.) Okta, Inc.

Pundit. (2018). Github. Retrieved from https://github.com/varvet/pundit

React. (2018). React docs. Retrieved from https://reactjs.org/docs/hello-world.html

React-authorization. (2017). Retrieved from https://github.com/ledsoft/react-authorization

React-router role authorization. (2017). Bartłomiej Dybowski. Retrieved from https://github.com/burczu/react-router-role-authorization

Reschke, J. (2015). The 'basic' authentication scheme. Retrieved from https://tools.ietf.org/html/rfc7617

Saltzer, J. H., & Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE, 63*(9), 1278-1308.

Sandhu, R. S., & Samarati, P. (1994). Access control: Principle and practice. *IEEE Communications Magazine, 32*(9), 40-48. 10.1109/35.312842

Schafer, S. M. (2010). *HTML, XHTML, and CSS bible* (5th ed.) Wiley Publishing.

Schlosser H. (2018). Technology trends 2018: Here are the top frameworks. Retrieved from https://jaxenter.com/technology-trends-2018-frameworks-144575.html

Smith, S., & Scott, A. (2016). Dependency injection in ASP.NET core. Retrieved from https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.0#fundamentals-dependency-injection

TypeScript Interfaces. (2019). Retrieved from https://www.typescriptlang.org/docs/handbook/interfaces.html

Van Hoff, A. (1997). The case for java as a programming language. *IEEE Internet Computing, 1*(1), 51-56. doi:10.1109/4236.585172

Venners, B. (1998). *The java virtual machine.* McGraw-Hill, New York.

Walther, S., Pasic, A., & Dykstra, T. (2008). Understanding models,views and controllers (C#). Retrieved from https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/overview/understanding-models-views-and-controllers-cs

Zakas, N. C. (2009). *Professional JavaScript for web developers* (2nd ed.). Indianapolis, IN: Wiley Pub.

Zhang, R. (2010). *Relation based access control*. Heidelberg: AKA.

Zlobin, G. (2013). *Learning python design patterns*. Birmingham: Packt Publishing.